

Computational Modeling

Twan Basten, Marc Geilen, Jeroen Voeten

Eindhoven University of Technology, Eindhoven, The Netherlands

Electronic Systems group, tue.nl/es

{a.a.basten,m.c.w.geilen,j.p.m.voeten}@tue.nl

April 12, 2023

Contents

| | |
|---|-----------|
| Computational Modeling | 1 |
| A Languages, Automata and Property Checking | 13 |
| A.1 Languages | 14 |
| A.2 Regular Languages | 19 |
| A.2.1 The class of regular languages | 19 |
| A.2.2 Finite automata | 23 |
| A.2.3 Expressiveness and closure properties | 28 |
| A.2.4 Regular-language representations | 31 |
| A.2.5 Decidability issues | 38 |
| A.3 Checking Regular Properties | 39 |
| A.4 ω -Languages | 44 |
| A.4.1 ω -Regular languages | 44 |
| A.4.2 Büchi automata | 46 |
| A.4.3 (Non)determinism in Büchi automata | 50 |
| A.4.4 Expressiveness and closure properties | 55 |
| A.4.5 Conversions | 56 |
| A.4.6 Decidability issues | 58 |
| A.4.7 Checking ω -regular properties | 60 |
| A.5 Linear-Time Temporal Logic | 66 |
| A.5.1 Propositional logic | 67 |
| A.5.2 LTL | 72 |
| A.5.3 Checking LTL properties | 77 |
| A.6 Answers to Exercises | 79 |
| Exercise A.1 (Words and languages). | 79 |
| Exercise A.2 (Operations on languages). | 79 |
| Exercise A.3 (Regular expressions). | 79 |
| Exercise A.4 (Creating regular expressions). | 80 |
| Exercise A.5 (Languages of finite automata). | 80 |
| Exercise A.6 (Creating automata models). | 81 |
| Exercise A.7 (Completing automata models). | 83 |
| Exercise A.8 (Complement automata). | 85 |
| Exercise A.9 (Pumping lemma). | 86 |
| Exercise A.10 (Pigeonhole principle). | 87 |

| | |
|---|------------|
| Exercise A.11 (Conversion from a regular expression to an NFA- ε). | 88 |
| Exercise A.12 (Conversion from an NFA- ε to a regular expression, state elimination). | 88 |
| Exercise A.13 (Conversion from an NFA- ε to an NFA). | 90 |
| Exercise A.14 (Conversion from an NFA- ε to a DFA). | 90 |
| Exercise A.15 (Checking regular properties). | 92 |
| Exercise A.16 (ω -Regular languages, ω -regular expressions). | 93 |
| Exercise A.17 (ω -Regular expressions). | 94 |
| Exercise A.18 (Büchi automata). | 94 |
| Exercise A.19 (ω -regular expressions, Büchi automata). | 95 |
| Exercise A.20 (ω -regular expressions, Büchi automata). | 95 |
| Exercise A.21 (Finite automata, Büchi automata). | 96 |
| Exercise A.22 (NB/FA, DB/FA, and limits). | 96 |
| Exercise A.23 (NB/FA, DB/FA, and limits). | 97 |
| Exercise A.24 (Complement of Büchi automata?). | 97 |
| Exercise A.25 (Non- ω -regular language). | 98 |
| Exercise A.26 (From ω -regular expression to NBA). | 98 |
| Exercise A.27 (From NBA to ω -regular expression). | 100 |
| Exercise A.28 (Product automata). | 103 |
| Exercise A.29 (Checking ω -regular properties). | 103 |
| Exercises A.30, A.36, A.41 – Coffee machine. | 106 |
| Exercise A.30 (Checking ω -regular properties). | 106 |
| Exercise A.36 (Expansion). | 106 |
| Exercise A.41 (Checking properties). | 107 |
| Exercises A.31, A.39, A.43 – Arbiter. | 109 |
| Exercise A.31 (Propositional formula). | 109 |
| Exercise A.39 (Expansion, NBA). | 109 |
| Exercise A.43 (Checking properties). | 109 |
| Exercises A.32, A.34, A.37, A.40 – Analyzing a Mealy model. | 111 |
| Exercise A.32 (DBA system model, propositional specification). | 111 |
| Exercise A.34 (LTL property specification). | 112 |
| Exercise A.37 (Expansion, NBA). | 112 |
| Exercise A.40 (Property checking). | 112 |
| Exercises A.33, A.35, A.38, A.42 – Controller. | 113 |
| Exercise A.33 (DBA system model). | 113 |
| Exercise A.35 (LTL property specification). | 114 |
| Exercise A.38 (Expansion, NBA). | 114 |
| Exercise A.42 (Checking properties). | 114 |
| B Markov Modeling and Discrete-Event Simulation | 117 |
| B.1 Discrete-time stochastic processes | 119 |
| B.1.1 Probability distributions | 119 |
| B.1.2 Rewards and expected rewards | 120 |
| B.2 Markov chains | 122 |
| B.2.1 Independent identically distributed variables | 122 |

| | | |
|-------|--|-----|
| B.2.2 | Markovian property | 123 |
| B.3 | Transient analysis | 127 |
| B.3.1 | Transient analysis via matrix algebra | 127 |
| B.3.2 | Transient analysis via paths | 129 |
| B.4 | State classification | 132 |
| B.4.1 | Communicating states | 132 |
| B.4.2 | Recurrent and transient states | 134 |
| B.4.3 | Periodic and aperiodic states | 136 |
| B.5 | Hitting probabilities | 139 |
| B.5.1 | Probability to hit a state | 139 |
| B.5.2 | Expected cumulative reward until hitting a state | 142 |
| B.5.3 | Probability to hit a set of states | 144 |
| B.5.4 | Expected cumulative reward until hitting a set of states | 146 |
| B.6 | Long-run analysis | 148 |
| B.6.1 | Ergodic unichains | 148 |
| B.6.2 | Non-ergodic unichains | 151 |
| B.6.3 | General Markov chains | 155 |
| B.7 | Discrete-event simulation | 159 |
| B.7.1 | Basic estimation theory | 160 |
| B.7.2 | Estimating transient properties | 164 |
| B.7.3 | Estimating long-run properties | 166 |
| B.8 | Answers to Exercises | 170 |
| | Exercise B.1 (Wealthy gambler - expected reward). | 170 |
| | Exercise B.2 (Time-slotted Ethernetwork - throughput). | 170 |
| | Exercise B.3 (Expected reward - computation). | 170 |
| | Exercise B.4 (Transition diagram to matrix). | 171 |
| | Exercise B.5 (Matrix to transition diagram). | 171 |
| | Exercise B.6 (Markov chains - dependent and non-identically distributed variables). | 171 |
| | Exercise B.7 (Gambler's ruin - probability distributions). | 172 |
| | Exercise B.8 (Markov chains - independent identically distributed variables). | 172 |
| | Exercise B.9 (Probability distributions via matrix algebra). | 172 |
| | Exercise B.10 (N-step transition probabilities). | 173 |
| | Exercise B.11 (Gambler's ruin - n-step probabilities and expected reward). | 173 |
| | Exercise B.12 (Gambler's ruin - dependent and non-identically distributed variables). | 173 |
| | Exercise B.13 (Queue in time-slotted communication network - modeling and transient analysis). | 174 |
| | Exercise B.14 (Independent identically distributed variables as Markov chain). | 174 |
| | Exercise B.15 (Classes of a Markov chain). | 174 |
| | Exercise B.16 (State accessibility versus paths). | 175 |
| | Exercise B.17 (Communicating states - equivalence relation). | 175 |
| | Exercise B.18 (Recurrent versus transient classes). | 175 |
| | Exercise B.19 (Computing return probabilities through paths). | 175 |
| | Exercise B.20 (Periodic versus aperiodic classes). | 175 |

| | |
|---|------------|
| Exercise B.21 (Aperiodic states are eventually visited). | 176 |
| Exercise B.22 (Computing return probabilities through equations). | 176 |
| Exercise B.23 (Infinite closed classes are not necessarily recurrent). | 176 |
| Exercise B.24 (Hiccups in a video application). | 177 |
| Exercise B.25 (Hitting a state versus hitting a singleton state set). | 177 |
| Exercise B.26 (Hitting recurrent states with probability 1). | 177 |
| Exercise B.27 (Rover in a maze). | 178 |
| Exercise B.28 (Limiting matrix ergodic unichain). | 179 |
| Exercise B.29 (Video application - limiting distribution). | 179 |
| Exercise B.30 (Packet generator - generated load). | 180 |
| Exercise B.31 (Expected fraction of time spent in a state equals reciprocal of expected return time). | 180 |
| Exercise B.32 (Composition of two parallel packet generators - generated load). | 181 |
| Exercise B.33 (Computer system - throughput). | 182 |
| Exercise B.34 (Throughput of an Ethernetwork - confidence levels versus error bounds). | 182 |
| Exercise B.35 (Throughput of an Ethernetwork - required length of simulation sequence). | 182 |
| Exercise B.36 (Interval estimator of expected value). | 182 |
| Exercise B.37 (Confidence interval interpretation). | 183 |
| Exercise B.38 (Standard deviation - point estimator). | 183 |
| Exercise B.39 (Estimation transient distributions). | 184 |
| Exercise B.40 (Gambler's ruin - expected reward estimation). | 184 |
| Exercise B.41 (Gambler's ruin - converge rate central limit theorem). | 184 |
| Exercise B.42 (Estimation hitting probability - impact of maximal path length). | 185 |
| Exercise B.43 (Estimation expected cumulative reward until hit). | 185 |
| Exercise B.44 (Rover in a maze - estimation escape probability). | 185 |
| Exercise B.45 (Absolute error bound long-run expected average reward). | 186 |
| Exercise B.46 (Long-run expected average reward - confidence interval). | 186 |
| Exercise B.47 (Expected return time versus long-run expected average reward). | 187 |
| Exercise B.48 (Interval estimator long-run expected average reward). | 187 |
| Exercise B.49 (Estimation long-run expected fraction of time spent in a state). | 187 |
| Exercise B.50 (Estimation Cezàro limiting distribution of a non-unichain). | 187 |
| Exercise B.51 (Video application - long-run average buffer occupancy). | 188 |
| C Max-Plus Linear Systems, Dataflow Models and Performance Analysis | 189 |
| Introduction | 189 |
| C.1 Timed Synchronous Dataflow Models | 190 |
| C.2 Schedules and performance metrics | 200 |
| C.3 Max-Plus Algebra | 207 |
| C.4 Max-Plus Linear Systems | 211 |
| C.4.1 Event Sequences | 211 |
| C.4.2 Event Systems | 213 |
| C.4.3 Superposition | 218 |
| C.5 Impulse Response | 221 |

| | |
|---|-----|
| C.6 Max-Plus Linear Algebra | 225 |
| C.6.1 Definitions | 225 |
| C.6.2 Modelling Dataflow Graphs with Max-Plus Matrices | 228 |
| C.7 Converting Dataflow Graphs to Max-Plus Matrices | 230 |
| C.8 Monotonicity | 235 |
| C.9 The Eigenvalue Equation | 239 |
| C.10 Max-Plus Linear Systems State-Space Equations | 246 |
| C.10.1 Definition | 246 |
| C.10.2 Dataflow Graphs as Max-Plus-Linear Systems | 248 |
| C.11 Throughput Analysis | 254 |
| C.12 Latency Analysis | 258 |
| C.12.1 Definition | 258 |
| C.12.2 Computing Latency from the State-Space Representation | 262 |
| C.13 Stability | 268 |
| C.14 Answers to Exercises | 271 |
| Exercise C.1 (A wireless channel decoder – understanding dataflow) | 271 |
| Exercise C.2 (A producer-consumer pipeline – creating dataflow models) | 272 |
| Exercise C.3 (An image-based control system – understanding and creating dataflow models) | 272 |
| Exercise C.4 (A video decoder - multi-rate dataflow) | 275 |
| Exercise C.5 (An image-based control system – multi-rate dataflow) | 276 |
| Exercise C.6 (A wireless channel decoder – schedules, performance) | 278 |
| Exercise C.7 (A producer-consumer pipeline – schedules, performance) | 279 |
| Exercise C.8 (An image-based control system – schedules, performance) | 280 |
| Exercise C.9 (A wireless channel decoder – max-plus analysis) | 280 |
| Exercise C.10 (A wireless channel decoder – max-plus-linear index-invariant system) | 281 |
| Exercise C.11 (The manufacturing system – superposition) | 283 |
| Exercise C.12 (A wireless channel decoder – impulse responses) | 284 |
| Exercise C.13 (A producer-consumer pipeline – max-plus matrix equation) | 284 |
| Exercise C.14 (A producer-consumer pipeline – symbolic simulation) | 285 |
| Exercise C.15 (An image-based control pipeline – symbolic simulation) | 285 |
| Exercise C.16 (A producer-consumer pipeline – worst-case abstractions) | 286 |
| Exercise C.17 (An image-based control system – eigenvalue, eigenvector) | 287 |
| Exercise C.18 (A producer-consumer pipeline – eigenvalue, eigenvector) | 288 |
| Exercise C.19 (A wireless channel decoder – state-space model) | 289 |
| Exercise C.20 (A wireless channel decoder – throughput) | 291 |
| Exercise C.21 (An image-based control system – throughput) | 292 |
| Exercise C.22 (A producer-consumer pipeline – throughput) | 292 |
| Exercise C.23 (An image-based control system – latency) | 292 |
| Exercise C.24 (A wireless channel decoder – latency) | 293 |
| Exercise C.25 (A producer-consumer pipeline – latency) | 294 |
| Exercise C.26 (A wireless channel decoder – stability) | 294 |

| | |
|--------------------------------------|------------|
| Appendices | 297 |
| I Sets | 297 |
| II Relations and functions | 302 |

Computational Modeling

Model-driven design methods are essential to guarantee the proper functioning and the required performance of human-engineered, computer-controlled systems. These course notes introduce models of computation that enable the modeling of computer-controlled systems, with supporting functional analysis techniques, performance analysis techniques, and stochastic analysis techniques. The reader draws inspiration from design challenges in the embedded systems (ES) and cyber-physical systems (CPS) domains, but the covered models and analysis techniques are more broadly applicable. The notes cover the computational models of languages and automata, with accompanying property-checking techniques for functional analysis (Part A), discrete-time stochastic processes in the form of Markov Chains, with stochastic analysis and simulation support (Part B), and dataflow models and max-plus linear systems for performance analysis (Part C). The appendices contain some basic definitions and notations for sets, functions, and relations.

Learning objectives. After studying the course notes, the student will be able to

1. ***understand*** the relation between the covered modeling and analysis approaches, their use, and their limitations;
2. ***select*** and ***apply*** a fitting modeling approach and corresponding analysis for a problem at hand.

The course notes use colored boxes to assist the reader. Blue boxes provide definitions and introduce notations and concepts. Similarly, Green boxes provide examples and the yellow boxes contain exercises. A red box indicates an important result, like a proposition or theorem. Grey boxes provide background information that is not necessarily studied or understood to achieve the mentioned learning objectives.

Answers to exercises are given at the end of each part of the course notes. It is advised to first try exercises yourself, before looking up the answer. The course notes come with a workbench to practice with the material and to assist in solving exercises.

The Computational Modeling WorkBench (CMWB) is available at
<https://computationalmodeling.info/cmwb>.

The following exercises provide examples of the type of exercises that the reader that successfully studied these course notes should be able to answer. Answers to these exercises can be found at the end of this introduction.

Exercise 1 (A manufacturing system). Consider two machines M_1 and M_2 . The first machine performs two operations on a product, namely condition cn and drill dr . The second machine does a coating operation ct after which the product is dried and delivered, through a dd operation. After drilling, products are handed over from M_1 to M_2 via a (joint) handover operation ho .

A drill operation on M_1 may go wrong. In such cases, the product is beyond repair. It is then ejected from M_1 through an eject operation e (and hence not handed over to M_2). Also a coating operation on the second machine may not deliver the expected result in one go. So it may be repeated as often as necessary, where you may assume that at some point the coating is ok. Only with the proper coating, the product will be dried and delivered.

Machine M_1 will be ready to process a new product once the previous product has been handed over to machine M_2 or if that has been ejected. Machine M_2 will be ready to receive a new product (via a handover ho) as soon as the coating of the previous one is ok and the product is ready for drying and delivery. A handover of a next product from M_1 and the drying and delivery dd of the current product can be done in parallel. M_2 cannot start to coat a new product though before the previous product has been delivered.

1. Model the behaviors of the two machines as two finite automata, where only behaviors returning a machine to the empty state are considered final.

The two machines M_1 and M_2 are coupled together in system S . S works in a slotted fashion, where both M_1 and M_2 perform one operation in each slot, assuming they have an operation enabled; if no operation is enabled in any of the two machines, e.g., because there is no product being processed, then that machine stays idle during that slot. Machines do not unnecessarily stay idle and you may assume the availability of an unlimited supply of input products. A handover is an operation on both machines, that needs a slot, and can only be performed by the two machines simultaneously.

2. Model the behavior of the system S as a Büchi automaton.

We now want to verify the following property P : “System S will always again deliver a final product.”

3. Model property P as an LTL formula.
4. Verify whether or not S satisfies P . Clearly elaborate and motivate all steps of the verification process.

Finally, we are interested in quantitative performance of S .

5. What is the best-case delivery rate of S , in produced products per time slot? Clearly elaborate and motivate all the steps in your answer, including an explanation of the model used and the analysis performed to come to your answer.
6. Now consider machine M_2 in isolation again and assume that the coating operation needs to be repeated one in 20 times. What is the expected delivery rate of M_2 ? You may assume that M_2 can execute a handover to obtain a new product whenever it is ready to start processing a new product. Clearly elaborate and motivate all the steps in your answer, including an explanation of the model used and the analysis performed to come to your answer.

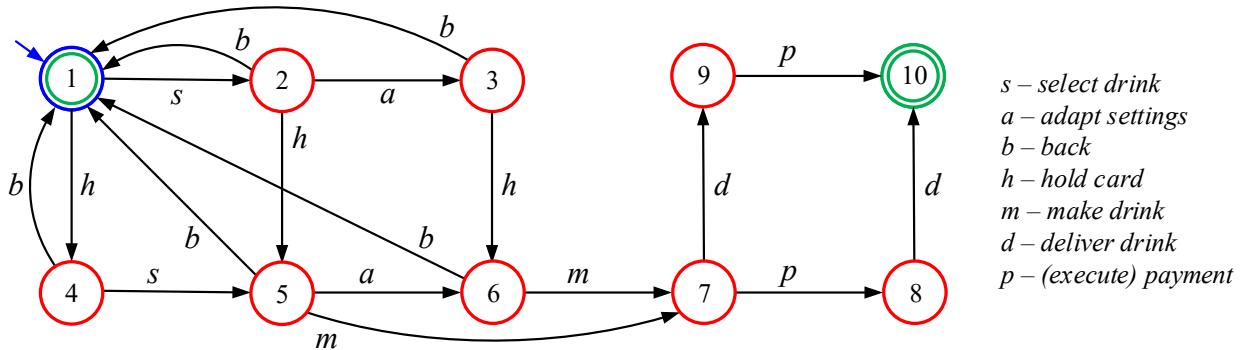


Figure 1: Deterministic finite automaton specifying a single sales of a drink.

Exercise 2 (A coffee machine). Consider a coffee machine of which the behavior of a single sales is specified by the automaton in Figure 1. The coffee machine is made of two units, an electronic unit for selection and payment, and a production unit, for making and delivering drinks. The s , a , b , h , and p actions are implemented in the electronic unit and the m and d actions in the production unit. A correct transaction has precisely one ‘deliver drink’ d and one ‘payment’ p action, in any order, whenever a drink has been made (‘make drink’ action m). A transaction that is aborted (by the user or a timeout via b actions) is also considered a correct behavior.

1. Of course, a real coffee machine is not supposed to deliver just a single drink. Provide a model of the coffee machine that specifies the continuous operation of the coffee machine and use it to verify whether or not the coffee machine continuously provides proper service in the form of correct transactions. Assume that the electronic unit is in use from the moment the selection of a drink is started till the payment of the drink has been executed or till the transaction is aborted; the production unit is in use from the moment a drink is made till the moment it is delivered. This implies that a new drink can be selected while the previous drink is still being made and delivered.
2. Now assume that making a drink takes 3 time units, that delivering it takes 2 time units, that a payment card needs to be held in front of the reader for 2

time units, and that a b action occurs automatically after 3 time units when nothing else happens. All other actions, including a user-initiated back action b , take 1 time unit. Delivering a drink and payment execution can be done in parallel, because the actions are processed by the two different units of the coffee machine. Reading a card, selection of drinks, and adaptation of settings can only be done in sequence, because they are performed by the electronic unit. What is the highest possible delivery rate of drinks if the machine is continuously used? What is the worst-case latency between the moment a drink is selected and the moment it is delivered?

3. Each time the coffee machine is used, it has a probability to fail. Assume that the probability that one of the two units of the machine fails is 1%. When one of the units is broken, the machine can no longer be used. A maintenance action makes sure that the machine becomes operational again. What is the mean time between failures, in terms of the number of drinks produced? Assume now that most of the time, with probability 998%, a maintenance action is performed properly. Sometimes however, with probability 2%, the machine is repaired using a quick fix. Each time a quick fix is carried out, the probability that the machine breaks with any subsequent use doubles. The coffee machine reaches the end of its economic life as soon as the mean time between failures drops below one hundred drinks. What is the expected economic lifetime (in terms of the number of drinks being served) of the machine?

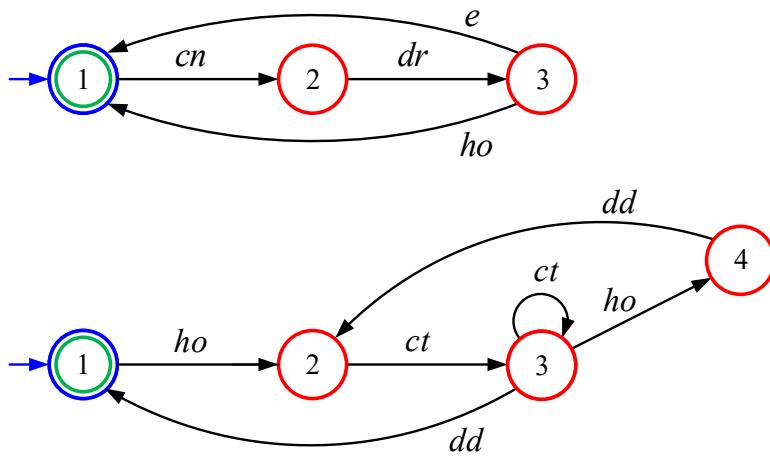
Acknowledgments

We thank Roy Meijer, Alireza Mohammadi, and Joep van Wanrooij for their contributions to the development of the practice material and the CMWB accompanying these course notes.

Answers to Exercises

Exercise 1 (A manufacturing system).

1. The two automata are as follows. For M_2 , one may observe that the flow for one product starts with a handover ho and is followed by one or more ct operations and a dd operation, after which the next product may be processed. This is captured through states 1 to 3, with the transitions between them. The ho of the next product may, however, come before the dd , which can be modeled by an extra state, state 4, and two extra transitions. The two different orderings of ho and dd for two subsequent products capture the parallelism.

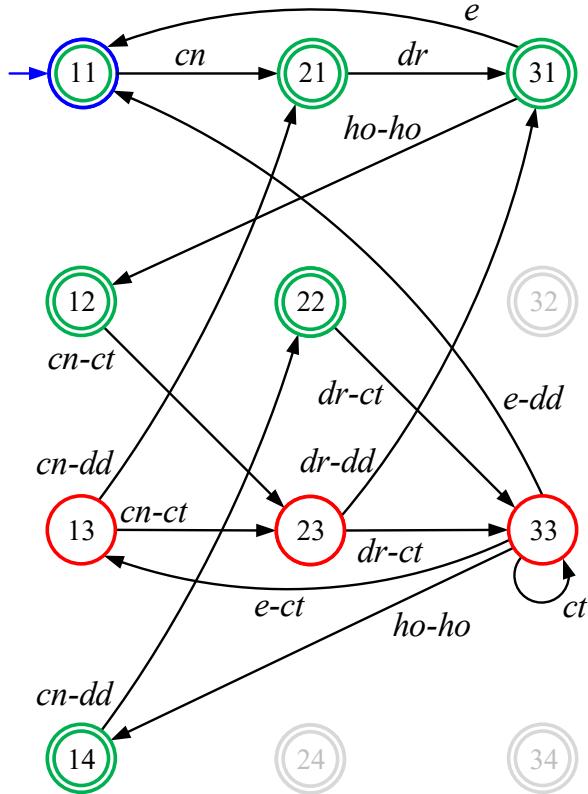


2. The key point to come to a model is to observe that the model can be seen as a product, similar to the product to compute language intersection. The two machines operate in lock step, under the assumptions and restrictions mentioned in the text. A correct model is given below, with some explanations as follows.

- The model has twelve states ij , corresponding to all combinations of states i and j of the original machine models. States that are not reachable from the initial state are greyed out, and their outgoing transitions are not shown.
- The model can be constructed by considering all possible outgoing transitions for each of the states, starting from the initial state 11, continuing till no new reachable states are found.
- Transitions are labeled with the operations from the two original machine models and new symbols $x-y$ denoting the combined execution of two actions x and y on the two machines in a single slot.
- Any pair of operations of the two machines may be done in parallel, except for the handover operations ho , which can only be paired together.
- Machine M_1 may operate in isolation at start-up, and any time later when drilling goes wrong and a product is ejected, while machine M_2 is waiting for a handover (so in states $i1$). Machine M_2 can only operate in isolation when the coating

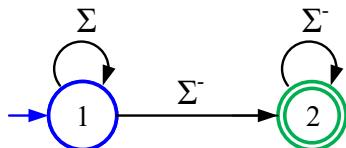
operation needs to be repeated, while the first machine is waiting for a handover (so in state 33).

- In principle, all behaviors specified by the automaton are acceptable, except that the text mentions that an infinite ct sequence cannot occur. Therefore, all states are final, except states $i3$. By not making these three states final, the two ct -labeled cycles in the automaton can only be taken finitely often for each product being processed. Note that correct models with fewer final states are possible.

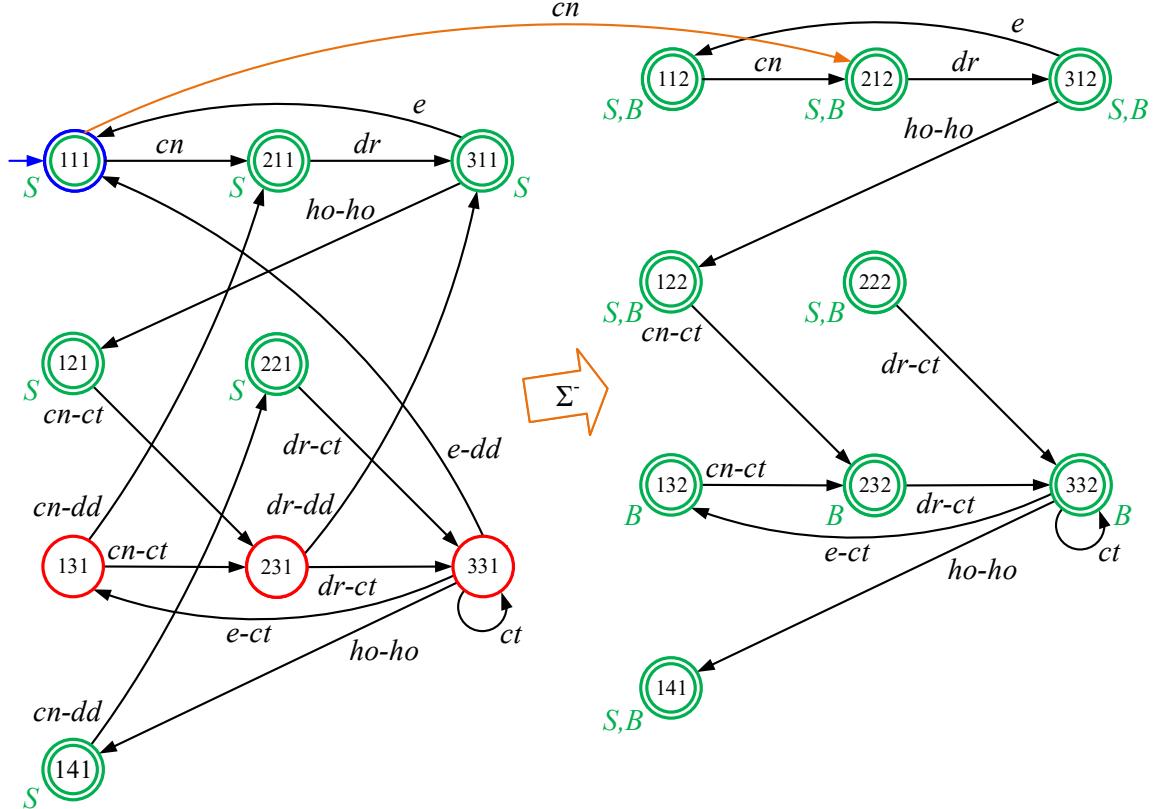


3. $\square \Diamond dd$. Note that $\square dd$ is not correct, because it does not allow for gaps in between two deliveries.
4. The verification consists of three steps.

First, we create an automaton of the bad behaviors, i.e., the behaviors where at some point in time no new dd is done. Let $D = \{cn-dd, dr-dd, e-dd\}$, that is all pairs of operations done by S in which a delivery is done. Let Σ be the alphabet of the automaton given for S earlier. Let $\Sigma^- = \Sigma \setminus D$ be the alphabet excluding all delivery operations. The following Büchi automaton then specifies the bad behaviors:



The second step is to create the product of the BA for S and the BA for the bad behaviors. This product is a GNBA. The following pictures this GNBA. It essentially has two copies of the BA for S , where any transitions from D are omitted from the second copy. Moreover, it is possible to go from state $ij1$ to $kl2$ with symbol s if and only if that same transition exists between $ij2$ and $kl2$. The latter is sketched in orange in the figure. The GNBA has two final-state sets, labeled S and B , corresponding to the final states of the two original BA.

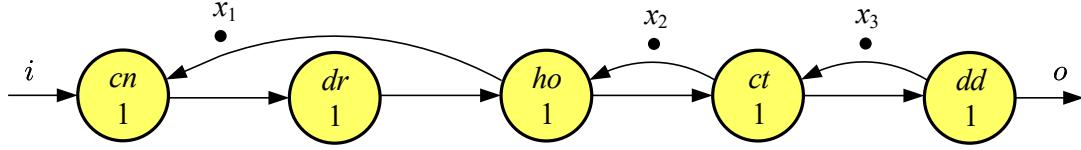


The final step is to check for emptiness of the language of the product GNBA. It is clear from the figure that it is possible to reach the cycle through states 112, 212, and 312. Since this cycle passes both final-state sets, the language of the GNBA is not empty. And hence the property is not satisfied.

Intuitively, this result may be understood by observing that the first machine may, from some point onward, no longer hand over products to machine M_2 due to persistent errors in the drilling. If this happens, the delivery of final products stops.

5. In the best case production scenario, drilling and coating do not fail. The performance of the system can be analyzed with a dataflow graph in which actors represent the individual operations. The two machines can be represented by constraints that ensure that the operations are executed in the right order and with the appropriate parallelism.

This leads to the following model.



The model has the following state matrix.

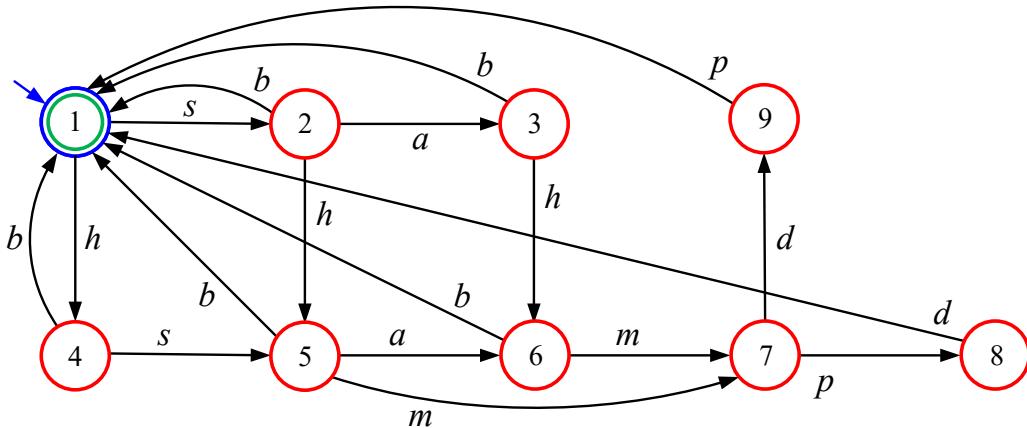
$$\mathbf{A} = \begin{bmatrix} 3 & 1 & -\infty \\ 4 & 2 & 1 \\ 5 & 3 & 2 \end{bmatrix}$$

The largest eigenvalue of this matrix is 3 (derived from the precedence graph) and, hence, the best-case throughput of the system is $1/3$, i.e., one product per three slots.

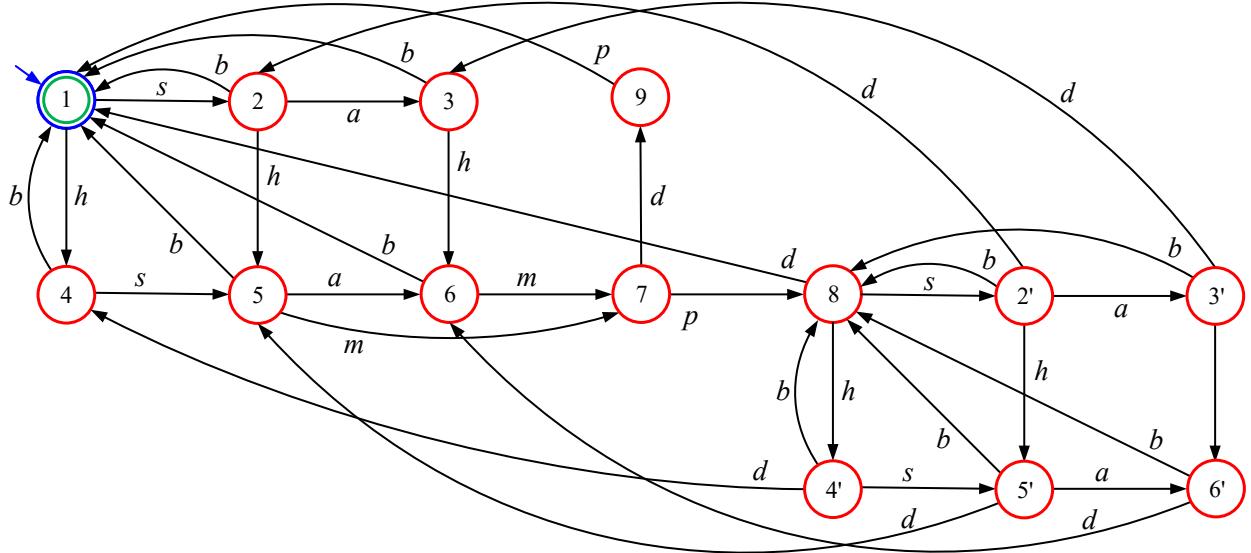
6. We can model M_2 as a Markov chain with state space $\{ho, ct, dd\}$ and the following transition probabilities: $P_{ho,ct} = 1$, $P_{ct,ct} = \frac{1}{20}$, $P_{ct,dd} = \frac{19}{20}$, $P_{dd,ho} = 1$. The delivery rate equals the long-run fraction of time the chain spends in state dd . It can be computed by solving the balance equations: $\pi_{ho} = \pi_{dd}$, $\pi_{ct} = \pi_{ho} + \frac{1}{20}\pi_{ct}$, $\pi_{dd} = \frac{19}{20}\pi_{ct}$. Solving yields $\pi_{ho} = \frac{19}{58}$, $\pi_{dd} = \frac{19}{58}$, $\pi_{ct} = \frac{20}{58}$. Hence, the delivery rate is $\frac{19}{58}$.

Exercise 2 (A coffee machine).

1. To capture continuous service, we need to create a Büchi automaton capturing the proper behavior. This can be done in two steps. In the first step, we observe that states 1 and 10 can be merged, creating a DBA that provides continuous service, but *without the option* to already start a next transaction while the current transaction is still being completed.



To come to the final model, we can then observe that the s , a , b , and h actions on the electronic unit are already enabled for a new transaction once the payment for the ongoing transaction has been executed. This can be captured by ‘replicating’ states 1 to 6 starting from state 8. At any time, the current transaction can complete by delivering a drink, which takes the automaton from state i' to the corresponding state i , effectively remembering the actions that were already taken. Note that state 5' does not enable an m action, because this action is only enabled after delivering the drink.

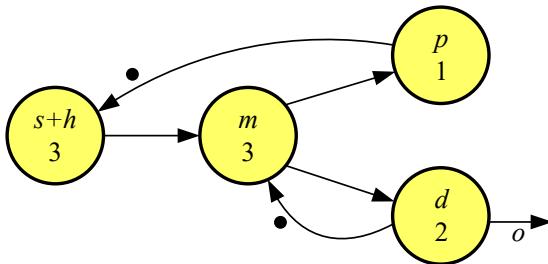


The proper-service property was introduced in Example A.28 (for a slightly different coffee machine). In LTL, it can be phrased as $\phi = \square(m \rightarrow \Diamond(((\neg m) \cup d) \wedge ((\neg m) \cup p)))$. The verification follows the approach outlined in Algorithm A.10 (Checking LTL properties) and Exercise A.41. Doing this verification manually is tedious. Instead, using the CMWB, the bad behaviors, specified as $\neg\phi$, can be captured in the CMWB as follows:

```
ltl formula B = not G(m=>X(((not m)U d)and((not m)U p)))
alphabet {m,d,p,s,a,b,h}
```

This property can be converted to an NBA. The product of the resulting NBA and the above DBA model of the coffee machine can then be generated. Checking this product for emptiness yields the (expected) result that the coffee machine functions properly.

2. We first observe that a and b actions are not needed to make a drink and they only delay the delivery of drinks if executed. So these actions can be ignored when considering the maximum delivery rate. We can now create the following dataflow model that captures the timing of the coffee machine when operating at maximum rate.



It has actors for the p , m , and d actions, with the mentioned durations as execution time. p and d may execute in parallel. Given that s and h both need to be done, in sequence, but in any order, we may capture their combined execution with a single actor with an execution time of 3. In line with the use of the two units of the coffee

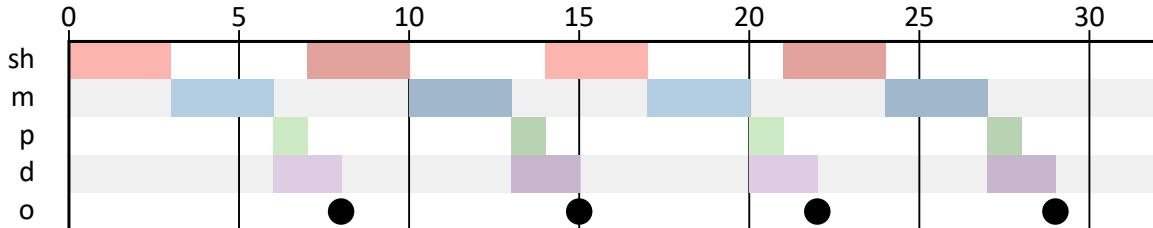
machine, the completion of p enables the next execution of s and h and the completion of d enables the next execution of m . Finally, the completion of d produces the drink, captured via output o .

We may enter this dataflow model in the CMWB and compute the throughput, which gives us the highest possible delivery rate. The analysis shows that this rate is $1/7$, i.e., one drink every seven time units, determined by the cycle through actors $s+h$, m , and p in the dataflow model. We can then compute the latency for period 7. The CMWB gives us an initial-state latency matrix $\begin{bmatrix} 8 & 5 \end{bmatrix}$. Assuming that the initial tokens in the model are available at time 0, i.e., both units of the coffee machine are operational at time 0, we obtain a latency of

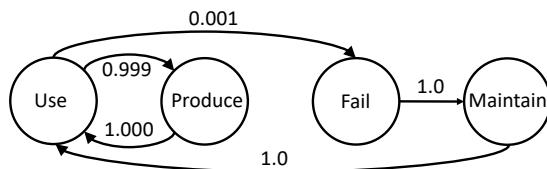
$$\begin{bmatrix} 8 & 5 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 8 .$$

That is, drinks are delivered (at most) 8 time units after starting the procedure to get a drink (when the machine is operating at the highest possible delivery rate, meaning that no a and b actions occur). This corresponds to the path through actors $s+h$, m , and d in the dataflow model. And although the latency is a worst-case delay between starting and completing buying a drink, it is not difficult to see that it always takes exactly eight time units to deliver a drink when not performing any a and b actions.

A Gantt chart illustrating these results is the following:

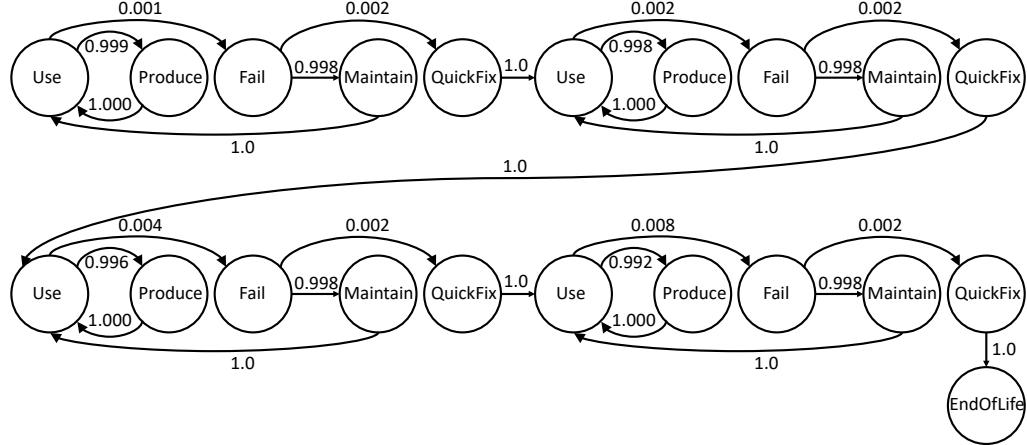


3. A transition diagram of a Markov-chain model is shown in the figure.



When in state *Use*, it can transit to state *Produce* with probability 0.999 in which a drink is delivered. It can fail with probability 0.001 and transition to the *Fail* state. After a drink has been delivered, the machine always returns to the *Use* state. After it has failed, a transition is made to the *Maintain* state and consequently to the *Use* state again. When we assume that reward 1 is obtained in state *Produce* and reward 0 in the other states, the mean time between failures is given by the expected cumulative reward until hitting *Fail* starting from state *Fail*. This result is given by $\frac{f_{\text{Fail},\text{Fail}}}{f_{\text{Fail},\text{Fail}}} = 999$. To address the question about the economic lifetime, we can extend the above model

with a *QuickFix* state and connect a number of model copies, each capturing the next doubling in failure rate:



Four copies are required, since the mean time between failures of a fifth copy (with failure rate 0.016) would have been 61.5 products (i.e., less than 100). Instead of this fifth copy a new state called *EndOfLife* is introduced. The expected life time is now given by $\frac{f_{Use, EndOfLife}}{f_{Use, EndOfLife}} = 935500$ products. All these results can be computed/verified through the CMWB.

A

Languages, Automata and Property Checking

Introduction

One of the most elementary and commonly used models to capture computation in systems design is the model of *languages* and *automata*. A language is a set of words over some given alphabet. Languages used in system design are so-called *formal languages* that have a precise mathematical definition, in contrast to natural languages that usually cannot be defined unambiguously. An *automaton* is a means both to specify a language and to recognize valid words of a language.

Precise models of patterns and behaviors in system design can be used to validate data, such as proper number representations, and to verify *properties* of the specified patterns or behaviors. As an example of the latter, it may be checked whether a resource arbiter always grants a resource request. If not, the arbiter may be unfair, or it may even deadlock. Properties may be specified through automata, but also through other means, such as in logics. A *logic* is a way of specifying languages that is often convenient to specify properties that a system should satisfy.

Words in a formal language may be finite or infinite. Finite words are often used to specify patterns, such as data input patterns that a given system can accept. Words may also specify the behaviors of a system, such as all possible sequences of actions a system may perform, of which the mentioned arbiter behavior is an example. Although in reality behaviors of systems are always finite, typically no bound is known on the behavior. The behavior of a cruise control system in a car, for example, or the behavior of a resource arbiter in an embedded system, depends on the duration of use of such a system. Infinite words form a good abstraction for the unbounded behavior of many systems that we see nowadays. One infinite word captures infinitely many finite behaviors. But even with this abstraction, languages are often infinite, in the sense that they contain infinitely many different words. Automata and logics are finite representations of languages that form the basis for specifying languages, for recognizing words in a language, and for property checking.

These course notes review the most important results from language theory, covering both languages consisting of finite words and those consisting of infinite words. The notes introduce several representations of languages, study their interrelations, and illustrate their use in pattern recognition and property checking. Good, classical, text books elaborating the

theory of formal languages with finite words are, for example, [15, 18, 19]. For further reading on languages with infinite words and property checking, see [3, 24], where [3] focuses on the property-checking perspective and [24] mostly on theoretical foundations. Also wikipedia contains much good-quality material on formal languages. The mentioned sources also served as a basis for these notes.

Learning objectives. After studying this part of the course notes, the student should be able to

- *read, understand, design* and *formally describe* languages, regular expressions, and finite automata;
- *argue* whether or not a language is regular, using the pumping lemma and/or the pigeonhole principle;
- *convert* between the various representations of regular languages (regular expressions, DFA, NFA, NFA- ε);
- *prove* whether or not a system of which the behavior is specified as a regular language satisfies a given regular property;
- *read, understand, design* and *formally describe* ω -languages, ω -regular expressions, and Büchi automata;
- *argue* whether or not a language is ω -regular or deterministically Büchi recognizable;
- *convert* between the various representations of ω -regular languages (ω -regular expressions, NBA, GNBA);
- *prove* whether or not a system of which the behavior is specified as an ω -regular language satisfies a given ω -regular property;
- *read, understand, and design* LTL properties;
- *convert* between LTL properties and (G)NBA;
- *prove* whether or not a system of which the behavior is specified as an ω -regular language satisfies a given LTL property.

A.1 Languages

A word over some alphabet of symbols is a string over that alphabet. In formal language theory, words over some alphabet can be used to define languages, just like in natural languages.

A *word* over some alphabet Σ is a string over that alphabet.

See the intermezzo on strings at the end of this section for some notations for strings (and hence words). We start by considering languages containing finite words only.

Definition A.1 (Language). Let set Σ be some alphabet of symbols. A (finite-word) *language* L over alphabet Σ is a subset of finite words from Σ^* . That is, $L \subseteq \Sigma^*$.

Example A.1 (Languages).

L1 Consider the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Language $L1 = \{0, 1, 4, 9, 16\}$ is the language of all squares under 20.

L2 Consider the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, E\}$. Language $L2$ is the language of all integer representations of the form $<[+/-] \text{ number } [E \text{ number}]>$, where the parts between square brackets are optional. For example, “+481”, “-42”, and “17E9” are all elements of $L2$, but “1.7E10” is not.

L3 Let Σ be the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Language^a $L3 = \{\sigma \in \Sigma^* \mid (\forall i : i \in \mathbb{N} : \sigma(i) = 0 \Rightarrow (\exists j : j \in \mathbb{N} \wedge j > i : \sigma(j) = 1))\}$ is the language of all natural-number representations in which every 0 is followed at some point by a 1. For example, “1201” and “1400315” are elements of $L3$; “3210” is not.

L4 Consider the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, =\}$. Language $L4$ is the language of all natural-number representations, well-formed additions and well-formed addition equations:

- the word “0” is in $L4$;
- every nonempty word that does not contain “+” or “=” and does not start with “0” is in $L4$;
- a word containing “+” but not “=”, with “+” separating two valid words of $L4$ is in $L4$;
- a word containing exactly one “=”, that separates two valid words of $L4$ is in $L4$;
- no other word is in $L4$ than those implied by the previous rules.

For example, “481”, “27+15”, “1+17 = 18”, and “481+18=42” are elements of $L4$, but “040+2” is not.

This last example shows clearly that languages specify the *syntax* of words, i.e., how words are composed of symbols, but not their *semantics*, i.e., what these words mean. That is why the descriptions of the languages refer to ‘representations’ of numbers and integers, instead of to numbers and integers directly.

^aThe formal definition of $L3$ uses the logic quantifiers \forall and \exists ; see the intermezzo below for some explanations on quantifiers and quantifier notations.

Exercise A.1 (Words and languages). Let Σ be the alphabet $\{0, 1\}$.

1. Consider language $L5 = \{\sigma \in \Sigma^* \mid (\forall i : i \in \mathbb{N} : \sigma(i) = 1 \Rightarrow \sigma(i + 1) = 0)\}$. Which of the words “00”, “0100”, “0110”, and “01010001” are elements of $L5$?
2. Consider language $L6 = \{\sigma \in \Sigma^* \mid (\forall i : i \in \mathbb{N} : \sigma(i) = 1 \Rightarrow \sigma(i + 1) = 1)\}$. Which of the words “00”, “0100”, “0110”, and “01010001” are elements of $L6$?
3. Consider language $L7 = \{\sigma \in \Sigma^* \mid (+i : i \in \mathbb{N} \wedge \sigma(i) = 1 : 1) \text{ is even}\}$. Which of the words ε , “00”, “0100”, “0110”, and “01010001” are elements of $L7$?
4. Precisely define the language $L8$ of (finite) words over Σ that contains all words with an equal number of 1s and 0s.

Answers to this exercise, and all following exercises, can be found in a separate section at the end of this part of the course notes. For electronic use, a hyperlink is provided as well.

See answer.

The examples of languages introduced in this section show that most languages have infinitely many elements. There are, for example, infinitely many numbers and infinitely many binary strings. Only language $L1$ is finite. Infinite languages cannot be explicitly enumerated, like language $L1$. Notations from set theory and predicate logic provide a basis for precisely defining such infinite languages. However, these definitions are not always easily understandable and they generally do not provide a basis to design algorithms for recognizing words of a language. To this end, the next sections look at various alternative ways to represent languages.

Intermezzo - Quantifiers

For quantifiers, we use the notation of [10] in which the bound variables are represented explicitly. To understand quantification, consider the standard multiplication operator. This operator has a number of properties. It is *commutative*: $x \cdot y = y \cdot x$; it is *associative*: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$; and it has a *unit element*: $1 \cdot x = x$. These three properties together imply that we can perform the multiplication of n elements in any order, and if n is zero, then the result is the unit element 1. A quantifier is also a commutative and associative operator with a unit element. Examples of well-known quantifiers are the logical quantifiers \forall (“for all”, universal quantification) and \exists (“exists” existential quantification). Given a quantifier Q and a non-empty list of variables \bar{x} , a quantification over \bar{x} is written in the following format: $(Q \bar{x} : D(\bar{x}) : E(\bar{x}))$, where D is a predicate that specifies the *domain* of values over which the variables in \bar{x} range and where $E(\bar{x})$ is the quantified *expression*. Any *binary* commutative and associative operator \oplus with a unit element can be generalized to a quantifier, also denoted \oplus , in a straightforward way. The unit element is needed to ensure that a quantification over an empty domain has a well-defined result.

Example (Quantifiers). Consider the binary $+$ on natural numbers. The sum of all natural numbers less than 10 can be written as follows using the summation quantifier $+$:

$$(+x : x \in \mathbb{N} \wedge x < 10 : x).$$

Note that commonly used alternative (but less precise) notations for this summation are $\sum_{x=0}^9 x$ and $\sum_{0 \leq x < 10} x$.

An important remark is moreover that one has to be careful with quantification over infinite domains. For example, adding infinitely many natural numbers is not necessarily well defined.

Exercise (Quantifiers).

1. What is $(+x : x \in \mathbb{N} \wedge x < 4 : x)$?
2. What is $(+x : x \in \mathbb{N} \wedge x < 0 : x)$?
3. What is $(\min i : i \in \mathbb{N} \wedge i \geq 4 : 2i)$?
4. What are the unit elements of universal quantification (\forall) an existential quantification (\exists)?

Answers

1. $(+x : x \in \mathbb{N} \wedge x < 4 : x) = 1 + 2 + 3 = 6$.
2. $(+x : x \in \mathbb{N} \wedge x < 0 : x) = 0$, because 0 is the unit element of summation and a quantification over an empty domain results in the unit element.
3. $(\min i : i \in \mathbb{N} \wedge i \geq 4 : 2i) = 2 \cdot 4 = 8$.
4. The unit element of universal quantification (\forall) is **true** and the unit element of existential quantification (\exists) is **false**.

Intermezzo - Strings

A definition of strings Let Σ be a set, called the alphabet of symbols. A *string* over the alphabet Σ is a sequence of symbols of Σ .

A *finite* string is a finite sequence of symbols. Formally, a *finite string of length n*, for some natural number $n \in \mathbb{N}$, over alphabet Σ is a total function $\sigma : \{0, \dots, n-1\} \rightarrow \Sigma$. The idea is that this function assigns a symbol to every *position* starting at position 0 and continuing to $n-1$. For every index j not in the domain of σ , $\sigma(j)$ is undefined.

The string of length zero is called the *empty string* and written ε .

Formally, $\varepsilon : \emptyset \rightarrow \Sigma$, for any alphabet Σ .

For the sake of readability, a string of positive length is usually written by juxtaposing the function values.

A string $\sigma = \{(0, a), (1, a), (2, b)\}$, for $a, b \in \Sigma$, is written aab . Note that $\sigma(2) = aab(2) = b$ and that $\sigma(4)$ and $aab(3)$ are both undefined.

In agreement with this convention, a single symbol is often identified with the string of length one consisting of that symbol. That is, a is sometimes interpreted as the symbol a and sometimes as the string of length one with symbol a on position 0.

The set of *all finite strings* of arbitrary length over alphabet Σ is denoted Σ^* .

An *infinite* string over alphabet Σ is an infinite sequence of symbols. It is represented by a total function $\sigma : \mathbb{N} \rightarrow \Sigma$. The set of natural numbers corresponds to an infinite sequence of positions and σ assigns a symbol from the alphabet to each position.

The set of *all infinite strings* over alphabet Σ is denoted Σ^ω . The set of all (finite and infinite) strings over Σ , denoted $\Sigma^{*\omega}$, is the union of Σ^* and Σ^ω .

Functions and relations on strings Function $|\cdot| : \Sigma^{*\omega} \rightarrow \mathbb{N} \cup \{\omega\}$ yields for each string its length: For each finite string $\sigma \in \Sigma^*$ of length n , $|\sigma| = n$ and, for each infinite string $\sigma \in \Sigma^\omega$, $|\sigma| = \omega$.

The length of string aab is 3; that is $|aab| = 3$. The length of the empty string ε is 0: $|\varepsilon| = 0$.

A symbol $a \in \Sigma$ is said to be an element of a string σ over Σ , denoted $a \in \sigma$, if and only if $a = \sigma(i)$ for some $i \in \mathbb{N}$ with $0 \leq i < |\sigma|$.

Symbols a and b are elements of string aab and symbol c is not; that is $a \in aab$ and $b \in aab$, but $c \notin aab$.

The concatenation of two strings $\sigma, \tau \in \Sigma^{*\omega}$, denoted $\sigma\tau$, is the sequence of length $|\sigma| + |\tau|$ defined as follows: For any $i \in \mathbb{N}$ with $0 \leq i < |\sigma|$, $\sigma\tau(i) = \sigma(i)$ and, for any $i \in \mathbb{N}$ with $|\sigma| \leq i < |\sigma| + |\tau|$, $\sigma\tau(i) = \tau(i - |\sigma|)$.

Concatenating string baa to string aab yields string $aabbbaa$. Concatenating a string with the empty string results in the original string, e.g., $baa\varepsilon = \varepsilon baa = baa$.

Note that, formally, concatenation can be seen as a function taking two strings as arguments and producing one string as a result. Further note the following consequence of the definition of concatenation.

If σ is infinite, then $\sigma\tau = \sigma$ for any τ .

Concatenating something to an infinite string doesn't change that infinite string because the concatenated part will never be reached when enumerating or traversing the symbols in the string starting from the beginning.

Finally, we introduce the prefix relation on strings. A string $\sigma \in \Sigma^{*\omega}$ is a *prefix* of some string $\tau \in \Sigma^{*\omega}$, denoted $\sigma \preceq \tau$, if and only if there is some string $\pi \in \Sigma^{*\omega}$ such that $\tau = \sigma\pi$.

For example, string aab is a prefix of $aabbaa$. Also ε , a , aa , $aabb$, $aabba$, and $aabbaa$ are prefixes of $aabbaa$; $aabbaa$ has no other prefixes.

A.2 Regular Languages

We start our investigation into finite representations of languages by introducing an important class of finite-word languages as defined in Definition A.1, namely the class of regular languages. It turns out that this class of languages has some convenient representations. We then look into various representations of regular languages, and the limits in expressiveness of this class of languages. Expressiveness in this context refers to the question which languages belong to the class of regular languages, and which languages do not belong to this class.

A.2.1 The class of regular languages

To facilitate the description of languages, it is possible to define *operations* on languages. These operations can be used to describe behaviors of systems in a compact way or to compose behaviors. The most common operations are the following.

Definition A.2 (Operations on languages). Let $L, L_0, L_1, L_2 \subseteq \Sigma^*$ be languages over some alphabet Σ .

Concatenation:

$$L_0 \cdot L_1 = \{\sigma\tau \mid \sigma \in L_0 \wedge \tau \in L_1\}$$

Note that concatenation is associative: $L_0 \cdot (L_1 \cdot L_2) = (L_0 \cdot L_1) \cdot L_2$.

Union:

$$L_0 + L_1 = L_0 \cup L_1$$

Note that union is commutative and associative: $L_0 + L_1 = L_1 + L_0$ and $L_0 + (L_1 + L_2) = (L_0 + L_1) + L_2$.

Intersection: Because languages are sets, the *intersection* of two languages is defined as $L_0 \cap L_1$. No special notation is introduced for language intersection. Intersection is commutative and associative.

Power:

$$\begin{aligned} L^0 &= \{\varepsilon\} \quad \text{and, for any } i \in \mathbb{N} \setminus \{0\}, \\ L^i &= L \cdot L^{i-1} \end{aligned}$$

Note that $\{\varepsilon\}$ is the language containing only the empty word ε .

Kleene closure:

$$L^* = (+ i : i \in \mathbb{N} \setminus \{0\} : L^i)$$

Note that this definition uses the quantifier $+$ corresponding to the binary union operation introduced above.

Transitive closure:

$$L^+ = (+ i : i \in \mathbb{N} \setminus \{0\} : L^i)$$

Note that $L^* = L^+ + \{\varepsilon\}$ (which does not mean that $\varepsilon \notin L^+$).

Complement:

$$\bar{L} = \Sigma^* \setminus L$$

We assume that concatenation binds stronger than union and intersection. Thus, $L_0 + L_1 \cdot L_2$ means $L_0 + (L_1 \cdot L_2)$ and not $(L_0 + L_1) \cdot L_2$; and $L_0 \cdot L_1 \cap L_2$ means $(L_0 \cdot L_1) \cap L_2$. Observe that the notation for the set of all finite words over alphabet Σ , Σ^* is consistent with the definition of the Kleene closure of the alphabet (which is itself a language).

Example A.2 (Operations on languages). Consider language $L2$ of Example A.1. Let $LD = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the language of digits. We can then specify the language LN of all numbers as $LD \cdot LD^*$. Moreover, we have language $\{+, -\}$ and elementary languages $\{E\}$ and $\{\varepsilon\}$, where the latter is the language containing only the empty word. $L2$ can then be specified as follows: $(\{\varepsilon\} + \{+, -\}) \cdot LN \cdot (\{\varepsilon\} + \{E\} \cdot LN)$. Note that the pattern of this expression follows the informal pattern already used in Example A.1.

Exercise A.2 (Operations on languages). Is concatenation of languages commutative? Motivate your answer.

See answer.

The operations on languages as introduced in Definition A.2 (Operations on languages) and the above example inspire a compact way of defining languages through so-called *regular expressions*. These regular expressions go hand-in-hand with the definition of the already mentioned class of languages, the *regular languages*.

Definition A.3 (Regular languages, regular expressions). Let Σ be some set of symbols.

- The empty set \emptyset is a regular language, denoted by the regular expression \emptyset ;
- the singleton language $\{\varepsilon\}$ containing only the empty word is a regular language, denoted by expression ε ;
- for every symbol $a \in \Sigma$, singleton language $\{a\}$ is a regular language, denoted by expression a ;
- for all regular languages L_0 and L_1 , represented by expressions α_0 and α_1 , the *union* $L_0 + L_1$ and *concatenation* $L_0 \cdot L_1$ are regular languages, represented by expressions $\alpha_0 + \alpha_1$ resp. $\alpha_0 \cdot \alpha_1$ (or $\alpha_0\alpha_1$ for short);
- for every regular language L , represented by expression α , *Kleene closure* or *Kleene star* L^* is a regular language, represented by α^* ;
- no other languages than the ones defined by the previous rules are regular languages (over alphabet Σ).

In other words, the set of regular languages is the smallest set of languages that contains primitive languages \emptyset , $\{\varepsilon\}$, and $\{a\}$ (for all $a \in \Sigma$) and is closed under union, concatenation, and Kleene star. The expressions defined by the above rules are called *regular expressions*. We use $\mathcal{L}(\alpha)$ to denote the language of regular expression α .

Closure. A set of objects is said to be *closed* under an operation if and only if the result of applying the operation on operands taken from the set is again an element of that set.

Note that the empty word ε and the individual symbols a of an alphabet are used in the above definition as expressions to represent the singleton languages $\{\varepsilon\}$ and $\{a\}$. This overloading of notation is convenient and usually does not cause confusion.

Example A.3 (Regular expressions). Consider languages $L1$ and $L2$ of Examples A.1 and A.2.

- Regular expression $0 + 1 + 4 + 9 + 1 \cdot 6$ defines $L1$.
- Regular expression $\alpha = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ defines language LD . Expression $\alpha \cdot \alpha^* (= \alpha\alpha^*)$ defines language LN . Expression $\beta = \varepsilon + ++ -$ (where $+$ is used as the union operation on regular expressions and $+$ denotes a symbol) defines language $\{\varepsilon, +, -\}$. Using these definitions, $\beta\alpha\alpha^*(\varepsilon + E\alpha\alpha^*)$ defines $L2$. Note that expression α^+ defined as $\alpha\alpha^*$ is often used to write repetitions of one or more elements from a language (though formally, α^+ is not a regular expression). This would simplify the definition of $L2$ to $\beta\alpha^+(\varepsilon + E\alpha^+)$.

Exercise A.3 (Regular expressions). Consider the regular expression for language L_2 given in Example A.3. Which of the following four regular expressions defines the same language L_2 ? If an expression defines a different language, then provide at least one word differentiating the languages.

1. $\beta\alpha\alpha^*(E\alpha\alpha^*)$;
2. $\beta\alpha\alpha^*(E\alpha\alpha)^*$;
3. $\beta\alpha\alpha^*(E\alpha\alpha^*)^*$;
4. $\beta\alpha\alpha^*(\emptyset + E\alpha\alpha^*)$;
5. Which of these regular expressions are equivalent to each other in the sense that they define the same language?

Consider the alphabet $\{a, b\}$. Which of the following pairs of regular expressions define the same language? If two expressions do not define the same language, provide at least one word differentiating the two languages.

6. $a(a^* + b^*)$ and $a(a + b)^*$;
7. $a(a^* + b)^*$ and $a(a + b)^*$.

See answer.

Exercise A.4 (Creating regular expressions).

1. Give a regular expression for language L_3 of Example A.1.
2. Give a regular expression for language L_4 of Example A.1.
3. Give a regular expression for language L_5 of Exercise A.1.
4. Give a regular expression for language L_6 of Exercise A.1.
5. Give a regular expression for language L_7 of Exercise A.1.
6. Can you give a regular expression for language L_8 of Exercise A.1? Motivate your answer.

See answer.

The definition of the class of regular languages leads to several interesting observations, and it raises several interesting questions as well. All finite languages are regular, because they can be explicitly enumerated using singleton languages containing single-symbol words, concatenation, and union. Also powers as defined in Definition A.2 are regular, because they can be defined using union and concatenation. But what about intersection and complement? Are the intersection and complement of regular languages regular again? Are there, in fact,

any languages (of finite words) that are not regular? And how do we recognize whether or not a word is an element of a language? We explore those questions in the next subsections.

A.2.2 Finite automata

An *automaton* is a mathematical device for characterizing languages. It can be used as a model or *specification* of a language, also called a language *generator* in such cases. But it is typically also used as a *recognizer*, i.e., a device that either accepts or rejects a given input word. An automaton can thus be used as a model specifying required patterns or system behaviors, or as a device capturing the patterns or behaviors of a realized system. An automaton essentially consists of *states* and *transitions* labeled with symbols from some alphabet between those states. A sequence of transitions specifies a word. All accepted sequences then specify a language. An automaton is a *finite* automaton if both the set of states and the alphabet of symbols are finite.

Definition A.4 (Finite automaton, NFA- ε). A finite automaton is defined by the 5-tuple $(Q, \Sigma, \Delta, Q_0, F)$ with

- Q a finite, non-empty, set of states,
- Σ a finite alphabet of input symbols,
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ the transition relation,
- $Q_0 \subseteq Q$ the non-empty set of initial states of the automaton, and
- $F \subseteq Q$ a set of final states.

This definition defines the class of so-called *nondeterministic finite automata* with ε moves or empty moves (NFA- ε). Note that transition relation Δ can alternatively be represented as a function $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ with the power set of states as codomain that gives for each state the set of states that can be reached by a given symbol or an ε move.

What is state? State is “All information from the past that is needed to determine the future.”

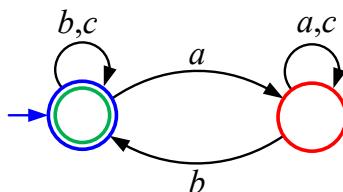


Figure A.1: A finite automaton.

Example A.4 (Finite automaton). A finite automaton can be graphically represented by means of a *state diagram*. This is a directed graph with vertices that represent the states and labeled edges for the transitions. By convention, an initial state is marked by a small incoming arrow and the final states are indicated by a double circle. For convenience, initial states are represented in blue, final states are represented in green and non-final states in red. Often, multiple edges are replaced by a single one labeled with a set of symbols (where curly braces are omitted). Figure A.1 visualizes the finite automaton $(Q = \{0, 1\}, \Sigma = \{a, b, c\}, \Delta = \{(0, b, 0), (0, c, 0), (0, a, 1), (1, a, 1), (1, c, 1), (1, b, 0)\}, Q_0 = \{0\}, F = \{0\})$. State identifiers are omitted in Figure A.1, as is often done in visualizations of automata.

An important subclass of automata, the class of *deterministic* automata, restricts the set of initial states to a single state, defines the possible state transitions as a function instead of a relation, and disallows ε moves. A deterministic automaton has a unique initial state and the transition relation defines for any combination of the current state and a symbol of the input alphabet at most one next state. *Nondeterminism* occurs when an automaton has more than one initial state, when a state has multiple outgoing transitions for the same symbol, or when it has ε moves. Another subclass of finite automata is obtained when nondeterminism is allowed but ε moves are excluded.

Definition A.5 (DFA, NFA). A *deterministic* finite automaton (**DFA**) is defined as in Definition A.4 but with Q_0 a singleton $\{q_0\}$ and Δ a (partial) function $\Delta : Q \times \Sigma \rightarrow Q$ that gives the unique next state that can be reached from a given state with a given symbol, if defined. A *nondeterministic* finite automaton (**NFA**) is defined as in Definition A.4 but with Δ a subset of $Q \times \Sigma \times Q$, or, alternatively, a function $\Delta : Q \times \Sigma \rightarrow 2^Q$.

The automaton of Figure A.1 is a deterministic one. Note that any DFA is also an NFA and any NFA is an NFA- ε .

Example A.5 (NFA- ε). Figure A.2 shows an example of a more elaborate state diagram; the automaton contains ε moves and is, hence, nondeterministic.

The purpose of an automaton is that it *defines* (or generates, recognizes, accepts) a formal language. The automaton of Example A.5, for example, defines the syntax of numbers in floating point notation, typically used in programming languages. Informally, a word is said to be accepted by an automaton if and only if there exists a path in the state graph of the automaton labeled by that word that leads from an initial state to a final state. The set of all words accepted by an automaton is called the language of that automaton. For readability, we first define the language of an automaton without ε moves.

Definition A.6 (Language of an NFA). Let $A = (Q, \Sigma, \Delta, Q_0, F)$ be an NFA. A word $\sigma \in \Sigma^*$ is *accepted* by A if and only if there is a *sequence of states* $\pi \in Q^*$ of length $|\sigma| + 1$, also called a *run* of A , such that $\pi(0) \in Q_0$, $\pi(|\sigma|) \in F$, and for all

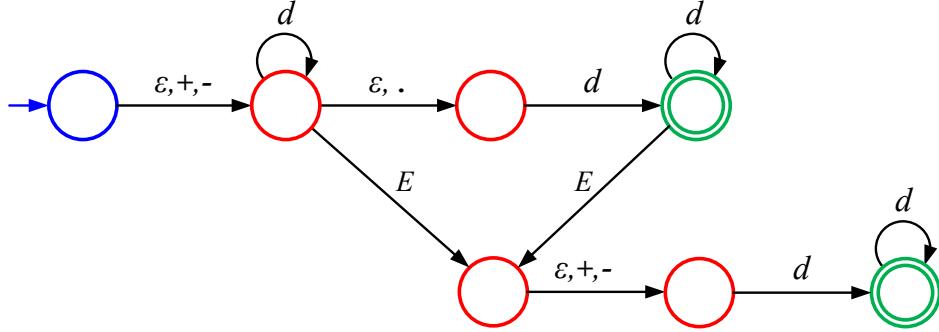


Figure A.2: An automaton for recognizing numbers represented in scientific notation (with ‘ d ’ being a shorthand notation for any digit from 0 to 9).

$i \in \mathbb{N}$ with $0 \leq i < |\sigma|$, $(\pi(i), \sigma(i), \pi(i+1)) \in \Delta$. The *language* of A , denoted $\mathcal{L}(A)$, is the set of all words accepted by A .

Example A.6. Consider the finite automaton of Example A.4 and visualized in Figure A.1. Run 00010 (a sequence of states) shows that the word $bcab$ is accepted.

The language accepted by an NFA- ε is defined in a way similar to Definition A.6, but the run that shows the acceptance of a word may include ε moves. Those ε moves do not contribute to the accepted word.

Let $A = (Q, \Sigma, \Delta, Q_0, F)$ be an NFA- ε . Let $\Rightarrow \subseteq Q^2$ be the relation defining **reachability via ε moves**: For any state $q \in Q$, $q \Rightarrow q$; for any $q, q' \in Q$, $q \Rightarrow q'$ if and only if there is a $q'' \in Q$ such that $(q, \varepsilon, q'') \in \Delta$ and $q'' \Rightarrow q'$. Overloading the notation \Rightarrow , we may now define a ternary relation $\Rightarrow \subseteq Q \times \Sigma \times Q$ capturing that a state is reachable from another state via a single-symbol word and ε moves: For any $q, q' \in Q$ and $a \in \Sigma$, $q \xrightarrow{a} q'$ if and only if there exist $p, p' \in Q$ such that $q \Rightarrow p$, $p \Rightarrow q'$, and $(p, a, p') \in \Delta$. Using these auxiliary notations, the language $\mathcal{L}(A)$ of NFA- ε A is defined following Definition A.6, by using \Rightarrow instead of Δ .

Definition A.7 (Language of an NFA- ε). A word $\sigma \in \Sigma^*$ is accepted by A if and only if there is a sequence of states $\pi \in Q^*$ (a run) of length $|\sigma| + 1$ such that $\pi(0) \in Q_0$, $\pi(|\sigma|) \in F$, and for all $i \in \mathbb{N}$ with $0 \leq i < |\sigma|$, $\pi(i) \xrightarrow{\sigma(i)} \pi(i+1)$.

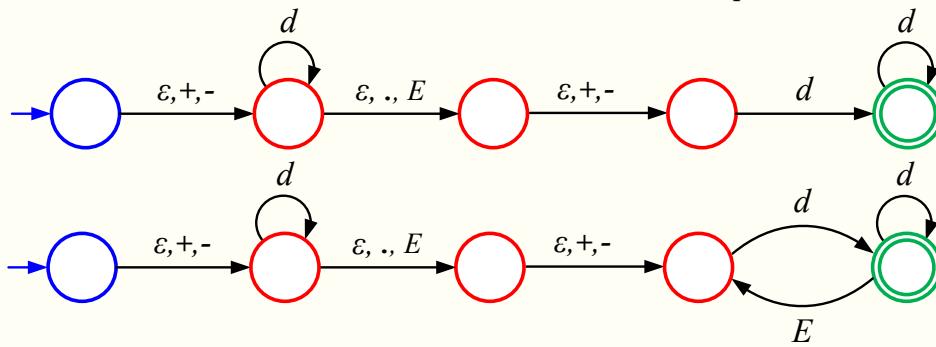
At this point, we have several mechanisms defining languages: regular expressions and finite automata. Finite automata moreover come in three different flavors. This raises the interesting question how the classes of languages defined by these four different means of specifying languages relate. One of the main results from the theory of finite automata and regular languages is the following. Proofs of this theorem can be found in the text books cited earlier.

Theorem A.1 (Regular languages). Regular expressions, DFA, NFA, and NFA- ε are equivalent with respect to the class of languages they define. They all define precisely the class of regular languages.

One may wonder why *nondeterminism* is useful. Nondeterminism in an automaton may make it difficult to recognize whether a word belongs to a given language or not. So it may complicate the use of an automaton as a recognizer. Nondeterminism may be useful though in *specifying* languages, to abstract from irrelevant details or to cope with unknown information such as unknown environment input or yet unknown implementation details. Nondeterminism, and ε moves in particular, moreover often allow more compact specifications of languages than achievable through DFA.

Exercise A.5 (Languages of finite automata).

1. Describe the language of the DFA of Example A.4, in words, using notations from set theory, and as a regular expression.
2. Consider two alternatives for the automaton of Example A.5:



These models are more compact than the automaton of Example A.5. But do they define the same language? If not, give counterexamples of words that are accepted by these automata but not by the automaton of Example A.5.

See answer.

Exercise A.6 (Creating automata models).

1. Give automata for languages L_1 and L_2 of Example A.1. Which of these automata are deterministic? For every nondeterministic automaton, give also a DFA.
2. Give automata for each of the languages of Exercise A.3. Which of these automata are deterministic? For every nondeterministic automaton, give also a DFA.
3. Give automata for each of the languages L_3 through L_7 of Example A.1 and Exercise A.1. Which of these automata are deterministic? For every nondeterministic automaton, give also a DFA.

4. Give a DFA for the automaton of Figure A.2.
5. Can you define a finite automaton for the language $L8$ of Exercise A.1? Motivate why (not)?

See answer.

For many practical purposes, it is convenient if the transition relation of an automaton is defined for all possible state/input-symbol combinations. Often, a system cannot control the inputs that are offered to it. If the transition relation of an automaton is not specified for all state/input-symbol combinations, the automaton is said to be incompletely specified.

Definition A.8 (Complete NFA, DFA). An NFA $A = (Q, \Sigma, \Delta, Q_0, F)$ without ε moves is said to be *complete* if $\Delta(q, a) \neq \emptyset$, for all $q \in Q, a \in \Sigma$. If A is a DFA, alternatively, A is complete if its transition function $\Delta : Q \times \Sigma \rightarrow Q$ is *total*.

Algorithm A.1 (Completing NFA, DFA). Any non-fully specified, incomplete, automaton without ε moves can be made complete by adding an extra, non-final state and turning all non-specified transitions into transitions to that new state. For the new state, we add a transition for each input symbol to that state itself (so that one can never again leave that state).

Example A.7 ((In-)complete automata). The automaton of Figure A.1 is complete. Figure A.2 shows an incomplete automaton. Figure A.3 shows an incomplete automaton and its completion.

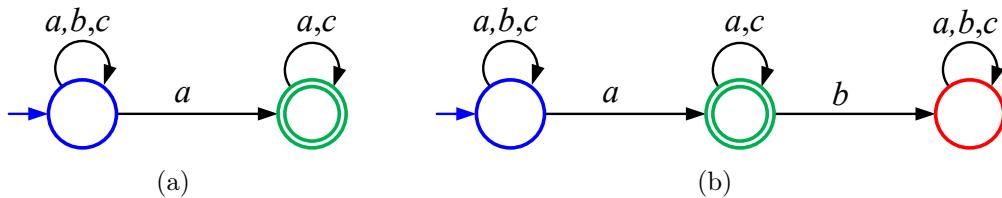


Figure A.3: A.3(a) An incomplete finite automaton; A.3(b) its completion.

The automaton resulting from the explained completion operation accepts the same language as the original automaton. The added state can be seen as an error state, from which there is no possibility to escape. This can be convenient when automata are used as recognizers. The completion operation can also be applied to NFA with ε moves. But the added transitions only add to the nondeterminism in the automaton. For automata with ε moves, it is best to first remove those ε moves or even convert to a DFA (see Section A.2.4 for how to do these conversions) and then apply completion.

Exercise A.7 (Completing automata models).

1. Reconsider all the DFA created in answers for Exercise A.6. Which of these is complete? For any incomplete DFA, give a complete DFA that accepts the same language.
2. Apply the procedure for completing an automaton to the NFA- ε of Figure A.2. Provide at least two example words in the language of the automaton for which the automaton in Figure A.2 has only one run (that is necessarily accepting), but for which the completed automaton has at least one additional run ending up in the added non-final state.

See answer.

A.2.3 Expressiveness and closure properties

Given Theorem A.1, it is interesting to study transformations between the various representations of regular languages. Before doing so though, we first investigate the *expressiveness* of the class of regular languages. That is, we first answer the question whether or not there are languages that are not regular. As may be expected from some of the exercises, there indeed are languages that are not regular. We therefore investigate which languages are regular and which are not, to build up an understanding of the concept of regularity.

Regular languages have been defined using several of the operations on languages defined in Definition A.2. However, not all operations from that definition were used. In particular, the intersection and complement of languages are not used in the definition. As already mentioned, this raises the questions whether the intersection of two regular languages is again a regular language and whether the complement of a regular language is regular. Or in other words, whether the class of regular languages is closed under intersection and complement.

Proposition A.1 (Closure properties of regular languages). The class of regular languages is closed under intersection and complement. That is, if L_0 , L_1 , and L are regular languages, then also $L_0 \cap L_1$ and \bar{L} are regular languages.

These closure properties can be proven using the automata representation of regular languages and a bit of set theory.

Definition A.9 (Complement of a DFA). Assume a *complete* DFA $A = (Q, \Sigma, \Delta, Q_0, F)$ accepting regular language L . Then the DFA \bar{A} with all final and non-final states interchanged, i.e., $\bar{A} = (Q, \Sigma, \Delta, Q_0, Q \setminus F)$, accepts the complement language \bar{L} . That is, $\mathcal{L}(\bar{A}) = \mathcal{L}(A)$.

Exercise A.8 (Complement automata).

1. Provide an example of an *incomplete* DFA A showing that the construction for complementing a complete DFA does not work for an incomplete DFA. That is, give an A such that $\mathcal{L}(\bar{A}) \neq \overline{\mathcal{L}(A)}$.
2. Provide an example of a complete NFA (so without ε moves) showing that the construction for complementing a complete DFA does not work for complete NFA.

See answer.

Now recall that $L_0 \cap L_1 = \overline{L_0} \cup \overline{L_1}$. Thus, closure under intersection follows from closure under union (part of the definition of regular languages) and closure under complement.

So the language complement and intersection do not yet yield non-regular languages. But they show that the class of regular languages is a well-defined class that is closed under all expected operations. The following theorem provides further insight in the regularity of languages.

Theorem A.2 (Pumping lemma). Let L be a regular language over alphabet Σ . Then, there is a positive natural number $p \in \mathbb{N} \setminus \{0\}$, the *pumping length*, such that for every word $\sigma \in L$ with $|\sigma| \geq p$, σ can be written as a concatenation of three words, $\sigma = \alpha\beta\gamma$, with $\alpha, \beta, \gamma \in \Sigma^*$, $\beta \neq \varepsilon$, and $|\alpha\beta| \leq p$, such that for all $i \in \mathbb{N}$, $\alpha\beta^i\gamma \in L$.

The pumping lemma states that any sufficiently long word in a regular language has a part that can be repeated arbitrarily often such that the resulting string is again a word in the language. The idea behind the pumping lemma may be understood by considering the earlier theorem stating that all regular expressions can be defined using a DFA. Since a DFA has finitely many states, any DFA recognizing a language with infinitely many words must contain a cycle. Otherwise, only finitely many words can be accepted. Such a cycle in a DFA can be repeated arbitrarily often when creating a run that accepts a word. Hence, the pumping lemma follows. This informal explanation is illustrated in Figure A.4.

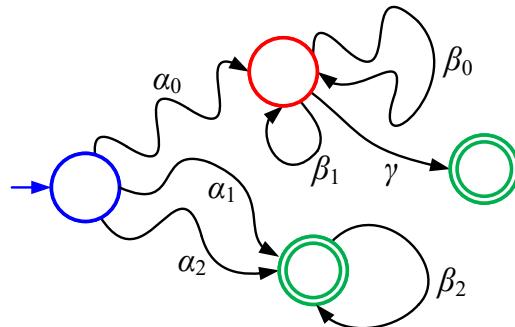


Figure A.4: An illustration of the pumping lemma. The shown automaton has a pumping length $p = \max\{|\alpha_0\beta_0|, |\alpha_0\beta_1|, |\alpha_1\beta_2|, |\alpha_2\beta_2|\}$.

Note that the informal illustration of the pumping lemma may seem to suggest that any

regular language has infinitely many words. This is not the case. The pumping lemma does not contradict the fact that all finite languages are regular. For any finite language, the pumping length can be chosen equal to the length of the longest word in the language plus one. The universal quantification over all words with a length of at least the pumping length is then a universal quantification over an empty domain, which is true by definition.

The pumping lemma can be used to prove non-regularity of languages, by reasoning towards a contradiction.

Example A.8 (Non-regular language, pumping lemma). Let $\Sigma = \{a, b\}$. Consider language $L9 = \{a^n b^n \mid n \in \mathbb{N}\}$ over alphabet Σ . Towards a contradiction, assume, in line with the pumping lemma, that natural number $p \in \mathbb{N} \setminus \{0\}$ is a valid pumping length for $L9$. Consider word $a^p b^p \in L9$. Since $|\sigma| \geq p$, it should be possible to rewrite σ as a concatenation of three words, $\alpha\beta\gamma$, with $\alpha, \beta, \gamma \in \Sigma^*$, $\beta \neq \varepsilon$, and $|\alpha\beta| \leq p$, such that for all $i \in \mathbb{N}$, $\alpha\beta^i\gamma \in L9$. This means that β must be of the form a^q for some positive natural number q at most p . But dropping this β from the word, or pumping it i times with $i \geq 2$, gives words that are no longer an element of $L9$. Hence, $L9$ cannot be regular.

Exercise A.9 (Pumping lemma).

1. Give for each of the languages $L2$ through $L7$ of Example A.1 and Exercise A.1 a valid pumping length. Give for each case also an example word and β that can be pumped.
2. Prove that language $L8$ of Exercise A.1 is indeed not regular.
3. The C programming language uses braces `{` and `}` to mark program segments. These braces need to occur in properly nested pairs. Modern editors have often built-in checks on the consistency of such nested pairs of braces, or other types of parentheses. Consider the language of properly nested pairs of braces. Is this language regular? If so, give a DFA accepting this language. If not, prove your claim using the pumping lemma.

See answer.

Another way of showing the non-regularity of a language is through the ‘pigeonhole principle’.

Pigeonhole principle. If $n \in \mathbb{N}$ objects are put in $k \in \mathbb{N} \setminus \{0\}$ boxes with $k < n$, then at least one box contains multiple objects.

Observe that, in an automaton, the only way to distinguish words that are acceptable and those that are not is through states. That is, only if two words lead to different states in the automaton, then these words, or extensions of them, can be differentiated in terms of acceptance. So, if a language contains infinitely many words that need to be distinguished, this cannot be done with finitely many states. In such a case, using the pigeonhole principle,

no finite automaton recognizing the language exists. So the language is not regular.

Example A.9 (Non-regular languages, pigeonhole principle). Consider again language $L_9 = \{a^n b^n \mid n \in \mathbb{N}\}$ of Example A.8. Towards a contradiction, assume the language is regular and a DFA exists that accepts L_9 . The DFA must be able to distinguish words a^n for all n because it must know how many b -s to recognize once the first b is received. For every $n \in \mathbb{N}$, word a^n can only lead to an accepting state in the DFA after processing a^n via a path corresponding precisely to b^n . Hence, every word a^n needs to lead to a different state in the DFA. Using the pigeonhole principle, this would need infinitely many states. Hence, the DFA cannot exist and L_9 cannot be regular.

This example shows that any language for which we need to ‘count’ symbols or substrings, without bound, is not regular.

Exercise A.10 (Pigeonhole principle).

1. Prove that language L_8 of Exercise A.1 is not regular using the pigeonhole principle.
2. Prove that the language of nested pairs of braces, defined in Exercise A.9, is not regular using the pigeonhole principle.

See answer.

Further reading - Non-regular languages. The theory of formal languages extends beyond the class of regular languages, introducing, for example, the classes of context-free languages and context-sensitive languages, the notion of grammars as a way to represent languages, and the Chomsky hierarchy of languages that clarifies the relations between some well-known classes of languages. The class of regular languages studied in these notes is the smallest, least expressive, class in the (original) Chomsky hierarchy. The interested reader is referred to, for example, [19] for more information.

A.2.4 Regular-language representations

Earlier, we have seen four different representations of regular languages. We have also seen that these four representations are equivalent, in the sense that they all specify the same class of languages (Theorem A.1). In this section, we investigate conversions between these representations. Note that these conversions also form the basis of the proof of Theorem A.1.

We start with the conversion of regular expressions to automata, NFA- ε , in particular. This conversion shows that each regular language can indeed be represented by an automaton. The use of nondeterminism and ε moves turns out to be very convenient in the conversion.

Algorithm A.2 (Conversion from a regular expression to an NFA- ε). Figure A.5 shows the conversion visually. The conversion follows the structure of Definition A.3 (Regular languages, regular expressions). The top part of the figure shows NFA for the primitive regular expressions \emptyset , ε , and a , for any a in the alphabet at hand. The other parts of the figure show the construction of NFA- ε for $\alpha\beta$, $\alpha + \beta$, and α^* , given automata recognizing languages for α and β . The union is obtained by simply taking the union of all states and transitions of the two constituent automata. The concatenation is obtained by adding ε moves from all final states of the α NFA to all initial states of the β automaton, and making the final states of α non-final and the initial states of β non-initial. The Kleene star is obtained by (i) creating a new initial state that is also final (to ensure that the empty word ε is accepted) with ε moves to the original initial states, (ii) adding ε moves from all the original final states to the new initial state (to allow iteration), and (iii) by making all the original initial and final states non-initial and non-final. The latter step is not strictly necessary, but it simplifies the automata resulting from the conversion.

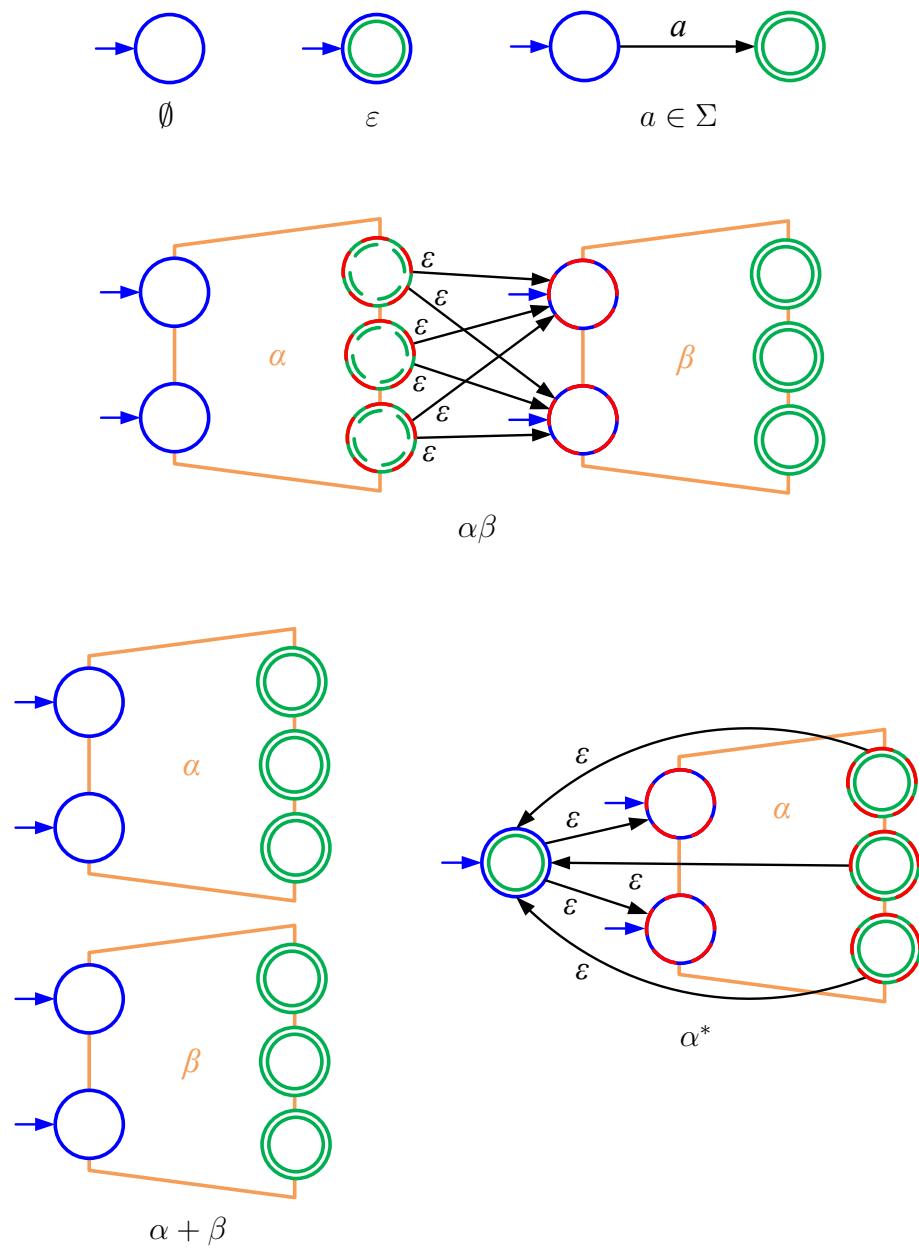
Exercise A.11 (Conversion from a regular expression to an NFA- ε). Consider regular expression $(a + b)^*aa^*$. Construct an NFA- ε accepting the same language as specified by the regular expression, rigorously following the conversion of Algorithm A.2 without simplification of intermediate NFA.

See answer.

The conversion from regular expressions to NFA- ε shows that every regular language can be represented by an automaton. However, it could still be the case that automata are more expressive, in the sense that they could capture languages that are not regular. This is not the case. The conversion from NFA- ε to regular expressions is a bit more involved than the conversion the other way around. We need some notation on single-step reachability in automata.

Notations - single-step reachability in automata. Let $A = (Q, \Sigma, \Delta, Q_0, F)$ be an NFA- ε . Recall from the alternative notation for Δ given at the end of Definition A.4 (Finite automaton) that, for any $q \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$, $\Delta(q, a)$ is the set of all states reachable from state q in a single step with symbol a or with ε in case $a = \varepsilon$, i.e., $\Delta(q, a) = \{q' \in Q \mid (q, a, q') \in \Delta\}$. For any $q \in Q$, let $\Delta(q) = \{q' \in Q \mid (\exists a : a \in \Sigma \cup \{\varepsilon\} : q' \in \Delta(q, a))\}$, i.e., $\Delta(q)$ is the set of all states reachable from q in a single step. We can generalize these notations to sets of states in the usual way by taking the union of the result sets. E.g., for any $Q' \subseteq Q$ and symbol $a \in \Sigma$, $\Delta(Q') = (\cup q : q \in Q' : \Delta(q))$ and $\Delta(Q', a) = (\cup q : q \in Q' : \Delta(q, a))$.

Algorithm A.3 (Conversion from an NFA- ε to a regular expression, state elimination). Let $A = (Q, \Sigma, \Delta, Q_0, F)$ be an NFA- ε . Assume $\mathcal{L}(A)$ is not one of the trivial languages \emptyset or $\{\varepsilon\}$ (for which the corresponding regular expressions are obvious). Assume that $Q = \{1, 2, \dots, n\}$ for some $n \in \mathbb{N} \setminus \{0, 1\}$; that is, A has at

Figure A.5: Conversion from regular expressions to NFA- ϵ .

least two states, numbered from 1 to n for some n . Further assume that $n - 1$ is the unique initial state of A ($Q_0 = \{n - 1\}$) without incoming transitions ($n - 1 \notin \Delta(Q)$). Moreover, assume that n is the unique final state of A ($F = \{n\}$) without outgoing transitions ($\Delta(n) = \emptyset$). Note that any automaton can easily be converted to this special form by introducing a new unique initial state with ε transitions to all the original initial states, creating a new unique final state with ε transitions to it from all the original final states, and by appropriately numbering the states. So we do not lose generality with these assumptions.

The conversion from NFA- ε A to a regular expression goes through a series of steps in which the states of the automaton, excluding the initial and final ones, are eliminated one by one (hence its name). The conversion uses an auxiliary notion of a *generalized* automaton in which transitions are labeled by regular expressions representing the (regular) language of all words corresponding to the (partial) runs from the source state of that transition to the result state of that transition.

The conversion of A to a regular expression follows the following steps:

1. For each (not necessarily distinct) pair of states $i, j \in Q$, if $j \in \Delta(i)$, replace all transitions between i and j in Δ with a single transition (i, α_{ij}, j) labeled with regular expression α_{ij} being the union of all the transition labels of the replaced transitions.
2. For k ranging from 1 to $n - 2$,
 - (a) for each pair of states $i, j \in Q \setminus \{k\}$ such that $k \in \Delta(i)$ and $j \in \Delta(k)$, add a transition if it does not yet exist, or update the transition if it does, with the regular expression label $\alpha_{ij} + \alpha_{ik}\alpha_{kk}^*\alpha_{kj}$;
 - (b) *eliminate* state k from Q and all its adjacent transitions from Δ .

In this second step, assume $\alpha_{mn} = \emptyset$ if $n \notin \Delta(m)$. Note that, from Definition A.2, it follows that $\emptyset + \alpha = \alpha + \emptyset = \alpha$, $\emptyset\alpha = \alpha\emptyset = \emptyset$, and $\emptyset^* = \varepsilon$.

After completion of these steps, α_{n-1n} is the sought for regular expression.

The order in which states are eliminated does not matter for the correctness of the conversion. The states were assumed to be (arbitrarily) numbered simply for convenience. Different elimination orders may yield (syntactically) different regular expressions though.

Example A.10 (Conversion from an NFA- ε to a regular expression, state elimination). Consider the automaton given at the top of Figure A.6. The automaton accepts the language of simple integer numbers, where the initial digit of a multi-digit number cannot be 0 and a plain 0 cannot be preceded with a + or -. Symbol p stands for a non-zero digit 1, 2, … 9. Symbol d can be any digit, as in earlier examples. Figure A.6 also gives an automaton satisfying the format constraints of Algorithm A.3. The conversion of the automaton to a regular expression goes through four

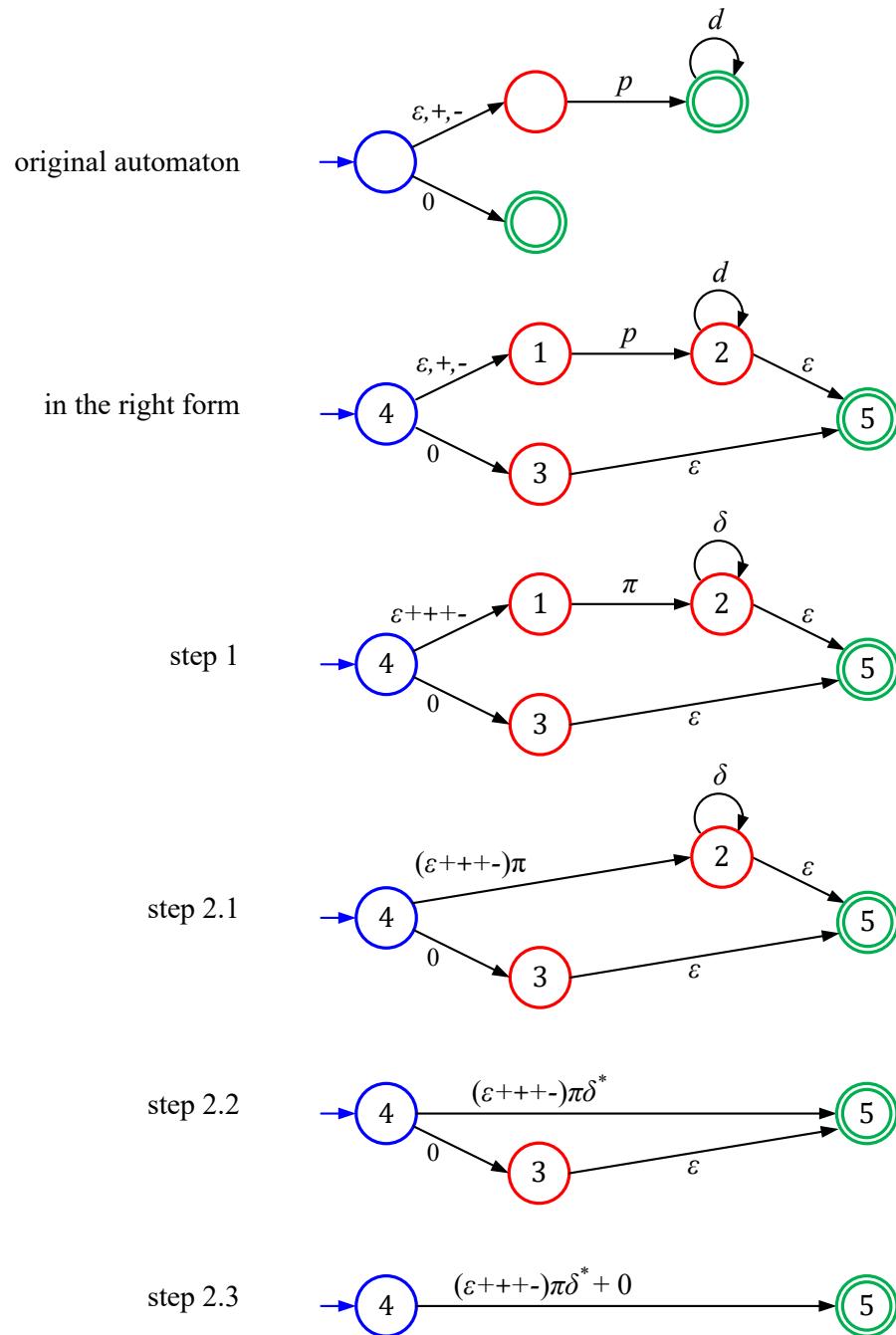


Figure A.6: Conversion from an NFA- ϵ to a regular expression through state elimination.

steps. Regular expressions π and δ represent the languages of non-zero digits and all ten digits, respectively. The last step of the conversion shows the regular expression $(\varepsilon + ++ -)\pi\delta^* + 0$ that accepts the same language as the original NFA- ε shown at the top of the figure.

Exercise A.12 (Conversion from an NFA- ε to a regular expression, state elimination).

1. Consider the automata for the languages $L2$ to $L7$ of Exercise A.6. Convert these automata to regular expressions using state elimination. Compare your answer to the regular expressions of Example A.3 and Exercise A.4.
2. Create a regular expression for the automaton of Figure A.2 using state elimination.

See answer.

The above two conversions show the expressive equivalence of regular expressions and NFA- ε , in line with Theorem A.1 (Regular languages). Theorem A.1 also states that the three variants of finite automata have equal expressiveness. It is clear that every DFA is an NFA, and that every NFA is an NFA- ε . In the remainder of this section, we give two more transformations, the first one providing a means to eliminate ε moves from an NFA- ε and the second one converting an NFA- ε into a DFA that accepts the same language. We need the following notation on reachability of states in automata.

Notations - ε -neighborhood. Let $A = (Q, \Sigma, \Delta, Q_0, F)$ be an NFA- ε . When defining the language of an NFA- ε (Definition A.7), we introduced the relation $\implies \subseteq Q^2$ denoting reachability through ε moves. Using this notation, we introduce, for any $q \in Q$, $E(q) = \{q' \in Q \mid q \implies q'\}$, i.e., the set of all states reachable from q with ε moves. This set is also called the **ε -neighborhood** of q . This notation is generalized to sets of states in the usual way: for any $Q' \subseteq Q$, $E(Q') = (\cup q : q \in Q' : E(q))$.

Algorithm A.4 (Conversion from an NFA- ε to an NFA). Let $A = (Q, \Sigma, \Delta, Q_0, F)$ be an NFA- ε .

1. For every state $q \in Q$ and state $q'(\neq q) \in E(q)$, i.e., every state q' in the ε -neighborhood of q ,
 - (a) if $(q', a, q'') \in \Delta$ for some $a \in \Sigma$ and $q'' \in Q$, add (q, a, q'') to Δ ;
 - (b) if $q' \in F$, i.e., if q' is final, then make q final;
2. remove all $(q, \varepsilon, q') \in \Delta$ from Δ , i.e., remove all ε moves.

Example A.11 (Conversion from an NFA- ε to an NFA). Consider again the automaton given at the top of Figure A.6. Figure A.7 shows an NFA without ε moves that accepts the same language. Note that the resulting automaton is in fact deterministic.

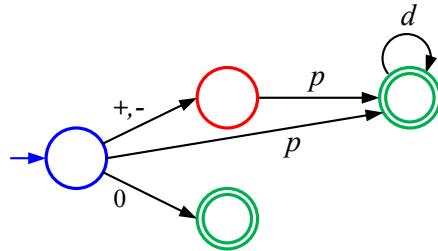


Figure A.7: Conversion from an NFA- ε to an NFA eliminating ε moves.

Exercise A.13 (Conversion from an NFA- ε to an NFA). Convert the automaton of Figure A.2 to an NFA without ε moves.
See answer.

Algorithm A.5 (Conversion from an NFA- ε to a DFA, subset construction). Let $A = (Q, \Sigma, \Delta, Q_0, F)$ be an NFA- ε . The key idea of the conversion is to imagine A being at any given moment in a set of states simultaneously instead of in a single state, namely all states that can be reached with the word processed so far. This basic idea gives the conversion its name. The DFA $(Q_d, \Sigma, \Delta_d, Q_{0d}, F_d)$ accepting the same language as A can be constructed in the following way:

1. Start with initial state $E(Q_0)$. That is, the set of states reachable from the initial states of the NFA through ε moves is chosen as the unique initial state of the DFA under construction. Formally, $Q_{0d} = E(Q_0)$ and $E(Q_0)$ is added as the first element of Q_d .
2. Recall the notation for single-step reachability introduced earlier in this section. For every $a \in \Sigma$, add $E(\Delta(Q_0, a))$ to Q_d and $(E(Q_0), a, E(\Delta(Q_0, a)))$ to Δ_d . That is, add the ε -neighborhood of all sets of states reachable in a single step via a specific symbol in the NFA as a state to the state set of the DFA, Q_d , and add the transition generating this state to the transition function of the DFA, Δ_d .
3. Continue adding $E(\Delta(Q', a))$ to Q_d and $(Q', a, E(\Delta(Q', a)))$ to Δ_d for any $a \in \Sigma$ and any state Q' newly added to Q_d until no new states occur anymore.
4. Add any $Q' \in Q_d$ to F_d if and only $Q' \cap F \neq \emptyset$. That is, any state $Q' \subseteq Q$ of the DFA that has a state q of the NFA that is final ($q \in F$) as one of its elements ($q \in Q'$) is a final state of the DFA ($Q' \in F_d$).

Example A.12 (Conversion from an NFA- ε to a DFA). Consider again the NFA- ε given at the top of Figure A.6. Use the state names of the equivalent automaton given in the same figure, i.e., the automaton has states $\{1, 2, 3, 4\}$ with initial states the singleton set $\{4\}$. Figure A.8 shows the DFA resulting from the subset construction that accepts the same language.

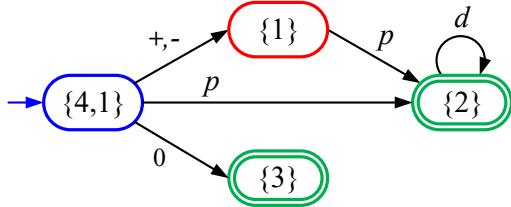


Figure A.8: Conversion from an NFA- ε to a DFA.

Exercise A.14 (Conversion from an NFA- ε to a DFA). Use the subset construction of Algorithm A.5 for the following conversions.

1. Convert the automaton in the second row of Figure A.6 to a DFA. Compare the result with the result of Example A.12.
2. Convert the automaton of Figure A.3(a) to an equivalent DFA.
3. Convert the two automata of Exercise A.5.2 to equivalent DFA.
4. Convert the automaton of Figure A.2 to a DFA. Compare the DFA with the DFA you gave in Exercise A.6.
5. Convert any NFA you created in Exercise A.6 for languages L_1 through L_7 to an equivalent DFA. Compare the result with the DFA you created yourself in that exercise.

See answer.

A.2.5 Decidability issues

Although it may seem strange at first, there are questions one can ask in mathematics that cannot be answered. We can sometimes even prove that a certain question is undecidable. A famous example is Turing's halting problem: Informally, it cannot be decided whether an arbitrary computer program will eventually halt executing. That is, there is no computer program that takes an arbitrary computer program as input and provides as a result whether or not that computer program terminates. The study of (un)decidability is intricately related with language theory and hence it is important to know of a class of languages whether certain properties can be decided for it. Basic decision problems posed in the realm of languages are the following: (i) Is a certain word an element of a given language? (ii) Is the given language

the empty language? Both problems are decidable for regular languages. The first question has obvious practical relevance. A DFA recognizing the regular language at hand can be used as a basis for deciding the question. But what about the second question? It turns out that together with the closure properties of the previous subsection, the fact that emptiness is decidable for regular languages allows us to conclude that inclusion and equivalence of two regular languages are also decidable. Emptiness of a regular language can be decided by checking whether or not an automaton representing that language accepts any words. That is, whether or not any of its final states are reachable.

Theorem A.3 (Decidability). Emptiness of a regular language is decidable. Since for regular languages L_0 and L_1 , $L_0 \subseteq L_1$ if and only if $L_0 \cap \bar{L}_1 = \emptyset$ and $L_0 = L_1$ if and only if $L_0 \subseteq L_1$ and $L_1 \subseteq L_0$, also inclusion and equivalence of regular languages are decidable.

These results have important practical relevance, as we will see in the next section.

Checking whether a language is empty. An automaton is a graph, with *finitely many* nodes and edges. By a depth-first search of that graph starting from each of the initial states, it can be decided whether any of the final states of the automaton are reachable. If so, the language is not empty. Otherwise, it is. A depth-first search of a graph can be efficiently implemented with a complexity that is linear in the size of the graph, the details depending on the used data structure.

A.3 Checking Regular Properties

Property checking. Let P be a language specifying the expected behavior of a system, the *Property*. Let S be the language describing all behaviors of the *System*. We would like to know whether the system satisfies the property, denoted $S \models P$. Intuitively, a system satisfies a property if and only if all behaviors of the system are in accordance with the property. When using languages as models for both systems and properties, property checking reduces to a check on language inclusion: $S \models P$ if and only if $S \subseteq P$. From the previous subsection, we know that $S \subseteq P$ if and only if $S \cap \bar{P} = \emptyset$. Note that \bar{P} specifies exactly all behaviors that are not in accordance with the desired property, the bad behaviors. So $S \cap \bar{P} = \emptyset$ captures the fact that system S does not have any bad behaviors.

Note that the above explanation of property checking is not restricted to regular languages. It refers to languages in general. Given a class of languages, an important question though is how to implement property checking.

The equation $S \cap \bar{P} = \emptyset$ forms the basis for an algorithm. In the previous subsection, we have already seen that checking emptiness of a regular language is decidable. It can be efficiently implemented via a depth-first search on an automaton-representation of the language. We have also seen how to complete a DFA and how to compute the complement of

a complete DFA. These operations are needed to compute the bad behaviors \bar{P} , given a DFA for the property. It remains to compute the intersection with the system behavior. Recall that $S \cap \bar{P} = \bar{S} \cup P$. However, this does not straightforwardly lead to an algorithm, because the union of automata leads to an NFA with ε moves that first needs to be turned into a complete DFA before it is possible to take the complement. This DFA may become very large. It is better to compute the intersection of two languages directly, without conversion to union. The key idea behind computing language intersection is to recognize a word for both languages simultaneously. Assuming automata models for two languages, their intersection can be computed by recognizing a word in both automata together, in lock step, one step at a time. Practically, one can create a so-called *product* automaton that implements this lock-step recognition.

Definition A.10 (Product NFA, language intersection). Let $A_1 = (Q_1, \Sigma, \Delta_1, Q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \Delta_2, Q_{02}, F_2)$ be two NFA with the same alphabet Σ . The product automaton $A_1 \times A_2 = (Q_1 \times Q_2, \Sigma, \Delta_\times, Q_{01} \times Q_{02}, F_1 \times F_2)$ with, for all $(q_1, q_2) \in Q_1 \times Q_2$ and $a \in \Sigma$, $\Delta_\times((q_1, q_2), a) = \Delta(q_1, a) \times \Delta(q_2, a)$. The *language* of the product automaton is the intersection of the languages of the original two automata: $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. Observe that transition relation Δ_\times captures precisely the lock-step execution of actions mentioned above. Only if both automata can make a step with a given symbol, the product automaton can make a step with that symbol.

Example A.13 (Product NFA). Consider alphabet $\Sigma = \{a, b\}$ and Figure A.9. The left part gives two automata, the operands of the product. The first automaton recognizes the language of all words containing an even number of a -s, or, in other words, words in which the number of a -s is divisible by 2. The second one recognizes all words in which the number of a -s is divisible by 3. The rightmost automaton is the product of these two operand automata. It has a set of six states, being the Cartesian product of the state sets of the operands. It has one initial state, namely the product state corresponding to the two initial states of the operand automata, and one final state, equal to the initial state, corresponding to the product of the two final states of the operand automata. The lock-step execution of actions is clearly visible. With a b , the product automaton does not change state, in line with the fact that both operand automata stay in the same state with a b . With an a , the product automaton changes to the state corresponding to the states that follow from a single step in the operand automata. The product automaton recognizes the language of all words in which the number of a -s is divisible by 6. This is indeed the intersection of the two languages of the two operand automata.

We now have all the ingredients for an algorithm that implements the property checking for regular languages.

Algorithm A.6 (Checking regular properties). Let P be a regular language specifying the property of interest. Let S be a regular language describing all system behaviors. It can be verified as follows whether $S \models P$:

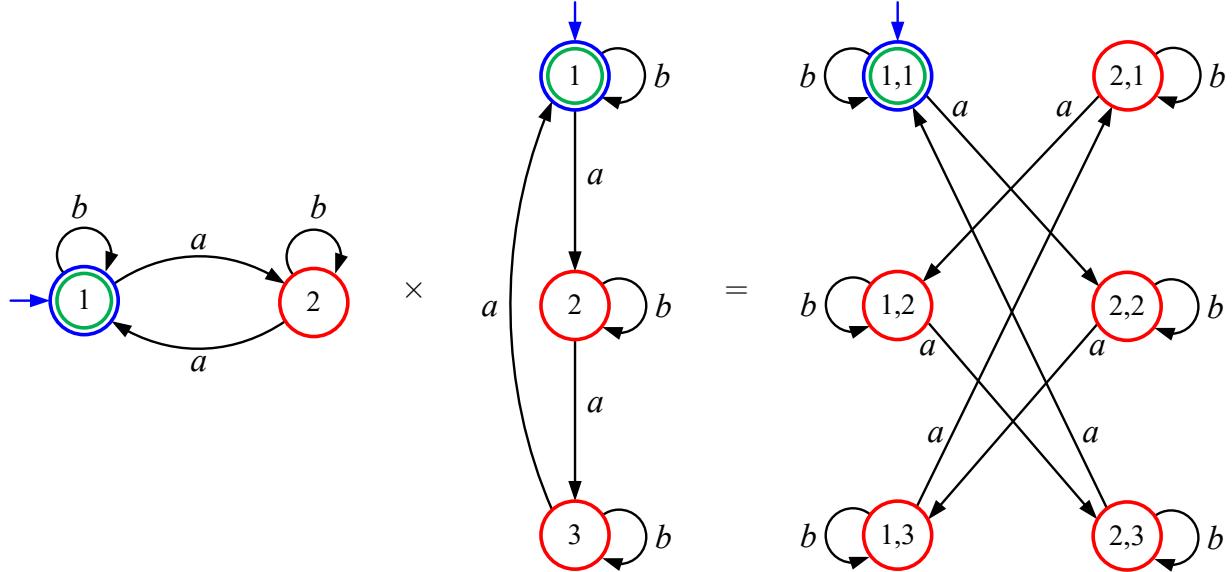


Figure A.9: The product of two NFA.

1. Create a complete DFA A_P for P ;
2. compute the complement automaton \bar{A}_P ;
3. create an NFA A_S for S ;
4. create the product automaton $A_S \times \bar{A}_P$;
5. check for emptiness of $\mathcal{L}(A_S \times \bar{A}_P)$.

If the language $\mathcal{L}(A_S \times \bar{A}_P) = \emptyset$, then $S \models P$, i.e., system S validates property P .

Note that the automaton for P has to be a complete DFA, because only then the complement can be efficiently computed. The system may be specified as an NFA, *without* ε moves, because the product automaton can be computed for NFA and also checking for emptiness can be done straightforwardly on an NFA. The conversions given in Section A.2.4 provide the means to obtain automata in the right format. Since the number of states of the product automaton is the product of the numbers of states of the two automata in the product, checking a property may still be time consuming, despite the efficient complement operation and emptiness check. Note that using NFA models for system behavior may improve efficiency of property checking, because an NFA may be more compact than a DFA recognizing the same language. Moreover, for any given DFA, it is possible to find a *minimal* DFA that accepts the same language with the smallest possible number of states (see e.g., [19], for a minimization algorithm). Exact DFA minimization is not necessarily efficient though, with a worst-case complexity that is exponential in the size of the DFA to be minimized. So heuristic alternatives exist that reduce the number of states and transitions of an automaton while preserving the accepted language, without guaranteeing a smallest result. Finally, observe

that in specific cases it may be efficient or convenient to directly start with a DFA of the bad behaviors, corresponding to the negation of the property of interest.

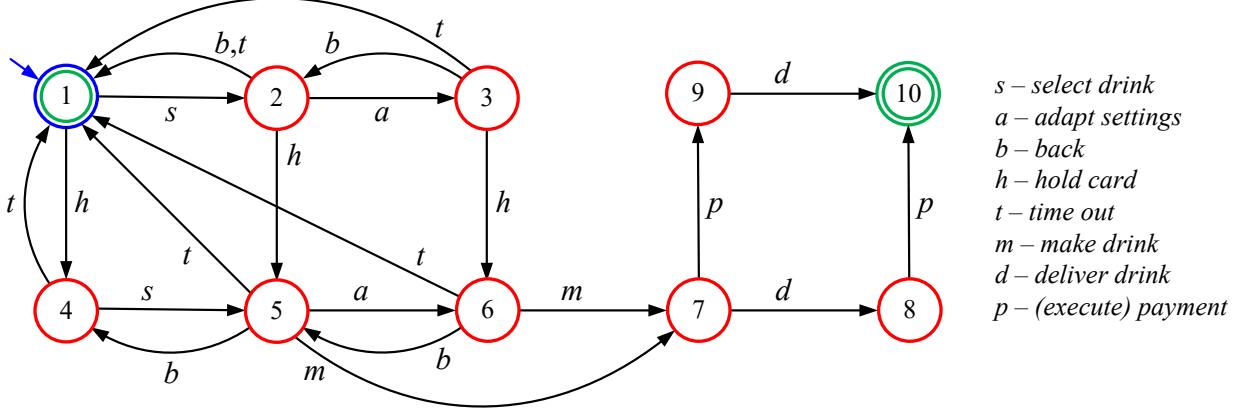


Figure A.10: DFA A_C specifying a coffee machine.

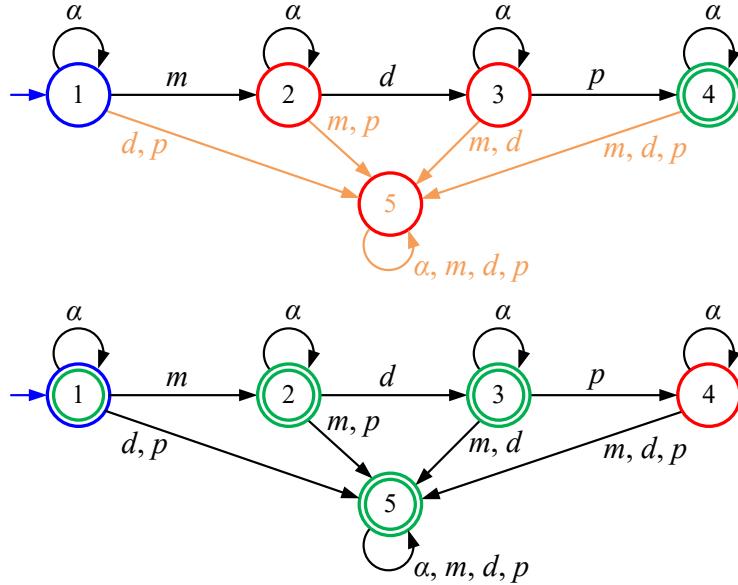
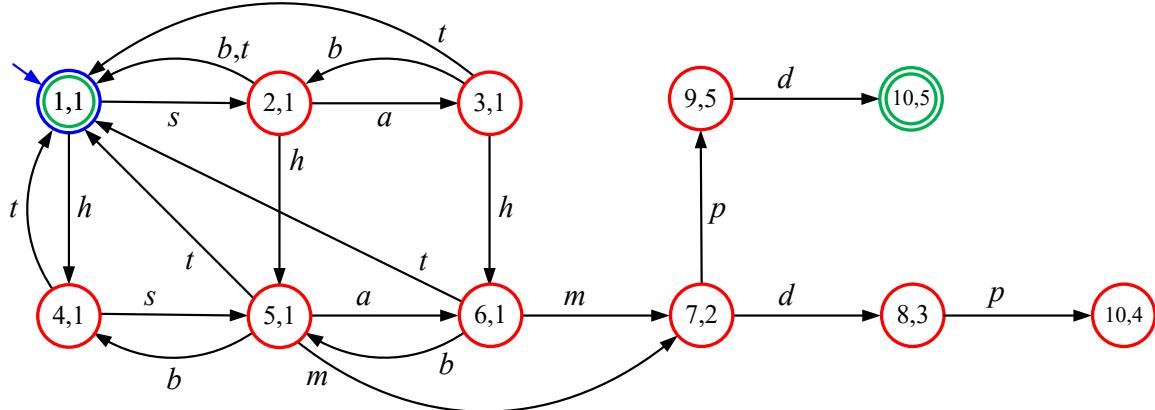


Figure A.11: DFA A_P for the regular property specifying correct sales of drinks (top) and its complement $A_{\bar{P}} = \bar{A}_P$ specifying all bad behaviors (bottom).

Example A.14 (Checking regular properties). Consider a coffee machine C , specified in Figure A.10. As a user of the coffee machine, we would like to verify that “every correct sale has exactly one ‘make drink’ m , one ‘deliver drink’ d , and one ‘payment’ p action, in the mentioned order. This property can be specified by the regular expression $\alpha^*ma^*da^*pa^*$, with $\alpha = s + a + b + h + t$ capturing all actions that are not of interest for the property. The top part of Figure A.11 shows a complete DFA specifying the property (Step 1 of Algorithm A.6), where regular expression α is used as an abbreviation on some of the transitions. The orange parts of the DFA are just for

Figure A.12: The product automaton $A_C \times A_{\bar{P}}$.

completion. The DFA without the orange parts also correctly specifies the property. The bottom part of Figure A.11 shows the complement of the complete DFA that defines the bad behaviors, \bar{P} (Step 2 of Algorithm A.6). We already have an automaton specifying the behavior of coffee machine C , Figure A.10 (Step 3 of Algorithm A.6). The next step is to compute the product automaton $A_C \times A_{\bar{P}}$, where A_C is the DFA of Figure A.10 and $A_{\bar{P}}$ is the bottom automaton of Figure A.11. The product automaton is shown in Figure A.12 (Step 4 of Algorithm A.6). The final step is an emptiness check. It is clear that the product automaton of Figure A.12 has two reachable final states, the initial state $(1,1)$ and state $(10,5)$. So language $\mathcal{L}(A_C \times A_{\bar{P}})$ is not empty. Therefore, the check on the validity of the property fails.

So, why does the check fail? First, the user may abort a transaction through back actions, or the coffee machine may abort a transaction after a time out, up to the point that a drink is made. This follows from the initial state also being final. Second, the machine may deliver the drink and process the payment in any order. The property requires the drink to be delivered before the payment is processed. Both are of course acceptable behaviors. Thus, we may conclude that our property was phrased too strictly. In the exercise below, we investigate the verification of an adapted property.

Exercise A.15 (Checking regular properties). Consider again the coffee machine of Example A.14.

1. Adapt the property P given in Example A.14 to allow transactions to be aborted by the user or by the coffee machine up to the point the drink is made, and to allow the processing of payment and drink delivery in arbitrary order. Show, following Algorithm A.6, that the adapted property is valid for the coffee machine.
2. We may try to cheat the coffee machine and get a free drink. Is that possible? Motivate your answer.

See answer.

A.4 ω -Languages

In many problems involving the modeling and analysis of system behavior, finite words are inadequate to describe this behavior: Many practical systems simply do not stop (or are not supposed to ever stop), meaning that their behavior is unbounded. In those cases, system behavior is best described by considering *infinite* sequences of events or actions. One such an infinite sequence captures infinitely many finite behaviors, namely all the finite prefixes of the infinite sequence. ω -languages are a special class of languages that deal with words of infinite length. In particular, ω -languages do not contain any finite-length words.

Definition A.11 (ω -Language). Let set Σ be some alphabet of symbols. An ω -language L over alphabet Σ is a subset of *infinite* words from Σ . That is, $L \subseteq \Sigma^\omega$.

As for languages of finite words, we need finite representations for ω -languages. We introduce both expressions and automata representations. The development follows the same lines as for finite-word languages, but with some important differences.

A.4.1 ω -Regular languages

To start, we need a means to construct infinite words from finite words.

Definition A.12 (Infinite concatenation). Let $K \subseteq \Sigma^*$ be a language (of finite words) for some non-empty alphabet Σ .

$$K^\omega = \{\sigma \in \Sigma^\omega \mid (\exists \sigma_0, \sigma_1, \dots : \sigma_i \in K \text{ for all } i \in \mathbb{N} : \sigma = \sigma_0\sigma_1\dots)\}.$$

The idea of K^ω is to generate an infinite word by concatenating an infinite selection of (finite, not necessarily different) words from K . Several of the operations on normal languages of Definition A.2 can be generalized to ω -languages.

Definition A.13 (Operations on ω -languages). Union, intersection, and complement, as defined in Definition A.2, carry over to ω -languages. Concatenation can be generalized to concatenate a language $K \subseteq \Sigma^*$ of finite words with a language $L \subseteq \Sigma^\omega$ of infinite words: $K \cdot L = \{\sigma\tau \mid \sigma \in K \wedge \tau \in L\}$.

Note that it is also possible to concatenate two ω -languages L_0 and L_1 . However, $L_0 \cdot L_1 = L_0$ if $L_0 \subseteq \Sigma^\omega$ is an ω -language. So such a concatenation is generally not very meaningful.

The operations introduced in the above definitions provide the basis for the following important class of ω -languages.

Definition A.14 (ω -Regular languages, ω -regular expressions). Let Σ be some non-empty set of symbols. The following rules define ω -regular languages and ω -regular expressions.

- For every regular language $K \subseteq \Sigma^*$ represented by regular expression α , *infinite concatenation* K^ω is an ω -regular language, represented by α^ω ;

- for every regular language $K \subseteq \Sigma^*$, represented by regular expression α , and ω -regular language L , represented by ω -regular expression β , concatenation $K \cdot L$ is an ω -regular language, represented by ω -regular expression $\alpha \cdot \beta$ (or $\alpha\beta$ for short);
- for all ω -regular languages L_0 and L_1 , represented by ω -regular expressions β_0 and β_1 , the union $L_0 + L_1$ is an ω -regular language, represented by ω -regular expression $\beta_0 + \beta_1$;
- no other languages than the ones defined by the previous rules are ω -regular languages (over alphabet Σ).

Note that these rules imply that any ω -regular expression α is of the form $(+ i : i \in \mathbb{N} \wedge 0 \leq i < n : \alpha_i \cdot \beta_i^\omega)$ for some $n \in \mathbb{N} \setminus \{0\}$ and α_i and β_i regular expressions for all $i \in \mathbb{N}$ with $0 \leq i < n$.

$\mathcal{L}^\omega(\alpha)$ denotes the language of ω -regular expression α .

It is common practice to drop the prefix ‘ ω -’ when referring to ω -regular languages and ω -regular expressions. It is usually clear from the context whether the terms refer to ω -languages or ordinary languages. Note that the empty language \emptyset is an ω -regular language, represented, e.g., by the ω -regular expression \emptyset^ω .

Example A.15 (ω -Regular languages, ω -regular expressions). Let Σ be the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Recall language $L3 = \{\sigma \in \Sigma^* \mid (\forall i : i \in \mathbb{N} : \sigma(i) = 0 \Rightarrow (\exists j : j \in \mathbb{N} \wedge j > i : \sigma(j) = 1))\}$, defining the language of all natural-number representations in which every 0 is followed at some point by a 1. It is straightforward to define an ω -language variant of this language: $L3^\omega = \{\sigma \in \Sigma^\omega \mid (\forall i : i \in \mathbb{N} : \sigma(i) = 0 \Rightarrow (\exists j : j \in \mathbb{N} \wedge j > i : \sigma(j) = 1))\}$. Since this language contains only infinite words, it no longer represents meaningful number representations. The language captures an important pattern though, namely the pattern that one type of event is always followed by a specific other type of event, a so-called *request-response* pattern. Such a pattern captures properties like patterns of traffic lights: any red light is eventually followed by green; or starvation freedom of a resource arbiter: if a party requests a resource of the arbiter, then it is eventually granted; or the property that any input to a system is eventually followed by an output.

To come to an ω -regular expression for $L3^\omega$, let $\alpha = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ be the language of digits; let $\beta = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ be the language of positive digits. Then, $(\alpha^* \cdot 1 + \beta)^\omega$ is an ω -regular expression specifying language $L3^\omega$.

Exercise A.16 (ω -Regular languages, ω -regular expressions).

1. Consider ω -language $L3^\omega = \{\sigma \in \Sigma^\omega \mid (\forall i : i \in \mathbb{N} : \sigma(i) = 0 \Rightarrow (\exists j : j \in \mathbb{N} \wedge j > i : \sigma(j) = 1))\}$ of Example A.15 with its regular expression $(\alpha^* \cdot 1 + \beta)^\omega$. Which of the following regular expressions is also correctly specifying $L3^\omega$?

- (a) $((\alpha^* \cdot 1)^* \cdot \beta^*)^\omega$;
- (b) $(\alpha^* \cdot 1 \cdot \beta)^\omega$;
- (c) $(\alpha^* \cdot 1)^* \cdot \beta^\omega$;
- (d) $(\alpha^* \cdot 1 \cdot \beta^*)^\omega + \beta^\omega$;
- (e) $(\alpha^* \cdot 1)^* \cdot \beta^\omega + \beta^\omega$.

Give at least one word distinguishing the languages for those expressions that do not specify $L3^\omega$.

2. Consider alphabet $\{a, b, c\}$. Give an ω -regular expression specifying the ω -regular language that contains all words in which every a is always followed by a b .
3. Consider the coffee machine of Example A.14. In that example, and the corresponding exercise, the proper sales of a drink was specified. Specify the property that the coffee machine provides continuous service of proper transactions, not limited to a single sales, as an ω -regular expression. You may assume that multiple transactions do not overlap.

See answer.

A.4.2 Büchi automata

For languages, we've introduced finite-state automata as a means to specify languages and as language recognizers. Similar representations, called ω -automata, exist for ω -languages. A well-known type of ω -automata are so-called Büchi automata. A (nondeterministic) Büchi automaton (NBA) is in fact simply an NFA as defined before, but with a different semantics that defines the language accepted by the automaton. Instead of accepting finite words, we need to accept infinite words. Whereas an NFA accepts a word if the word leads to a final state, an NBA accepts a(n infinite) word if it visits some ‘final’ state infinitely often. As for NFA, we may distinguish NBA with (NBA- ε) and without (NBA) ε moves.

The definition of the ω -language of an NBA- ε uses two auxiliary notations. Function inf on infinite words, defined below, gives all symbols occurring infinitely often in a word. For any symbol a , relation \xrightarrow{a} indicates that a state can be reached through a combination of ε moves and symbol a . This notation was already introduced to define the language accepted by an NFA- ε , Definition A.7.

Recall that ω represents the cardinality of the set of natural numbers \mathbb{N} . For any infinite string $\sigma \in \Sigma^\omega$, for some alphabet Σ , the set $\text{inf}(\sigma) = \{s \in \Sigma \mid |\{i \in \mathbb{N} \mid \sigma(i) = s\}| = \omega\}$ contains all elements of the alphabet that occur infinitely often in string σ .

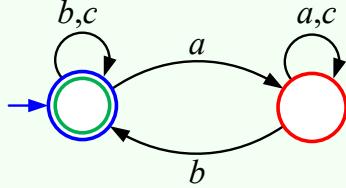
Definition A.15 (ω -Language of an NBA- ε). Let $B = (Q, \Sigma, \Delta, Q_0, F)$ be an NBA- ε . A word $\sigma \in \Sigma^\omega$ is *accepted* by B if and only if there is an infinite sequence

of states $\pi \in Q^\omega$, called a(n accepting) run, such that $\pi(0) \in Q_0$, for all $i \in \mathbb{N}$, $\pi(i) \xrightarrow{\sigma(i)} \pi(i+1)$, and $\inf(\pi) \cap F \neq \emptyset$. The ω -language of B , denoted $\mathcal{L}^\omega(B)$, is the set of all infinite words accepted by B .

Any ω -language that is recognizable by an NBA($-\varepsilon$) is referred to as a Büchi-recognizable language. We have the following important result, which states that the Büchi-recognizable languages are exactly the ω -regular languages.

Theorem A.4 (ω -Regular languages). ω -Regular expressions and NBA($-\varepsilon$) are equivalent with respect to the class of languages they define, namely the class of ω -regular languages.

Example A.16 (NBA). Recall the DFA of Figure A.1 over alphabet $\Sigma = \{a, b, c\}$.



Consider it as NBA B . What is the ω -language $\mathcal{L}^\omega(B)$ recognized by B ? Although the initial state also happens to be the final state, we cannot include the empty word because it is of finite length; the language can only include infinite words. All infinite words that are recognized must have an accepting run that passes through the final state infinitely many times. It should be obvious that no infinite word is accepted that stays in the self-loop on the non-final state. Clearly, the number of passes through that loop can only be finite, eventually ending by going via the edge labeled b back to the final state. We may therefore describe the infinite words in the language as having the characteristic that every a is always followed by a b ; if no a occurs at all, then the word may consist of any number of b -s or c -s in any combination. The language thus describes a request-response pattern as explained in Example A.15.

The language can also be described by an ω -regular expression, namely expression $(b + c + a \cdot (a + c)^* \cdot b)^\omega$. Since the initial state of the NBA is also a final state, the ω -regular expression corresponds to a single β_i in the pattern given in Definition A.14 (i.e., $n = 1$, $\alpha_0 = \varepsilon$, and $\beta_0 = b + c + a \cdot (a + c)^* \cdot b$).

(This is a good moment to reconsider Exercise A.16.2 and Exercise A.5.1.)

Let's now have a closer look at the acceptance conditions for an NBA. Figure A.13 may serve to illustrate the idea behind these acceptance conditions. Recall that an infinite word is accepted by an NBA if it corresponds to a run of the automaton that passes infinitely often through some final state. As the number of (final) states is finite, passing infinitely often through some final state implies that at least one final state is passed infinitely often. Thus an accepting run through an NBA can only be achieved if the run goes from some initial state to some final state through which it then passes infinitely often. Intuitively, each σ_{ij} in the figure corresponds to a path from some initial to some final state and each τ_{ik} corresponds

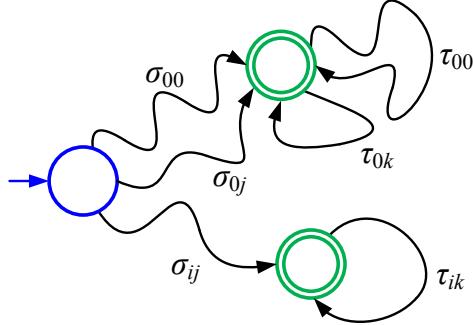


Figure A.13: An illustration of the acceptance conditions for an NBA.

to a cycle from that final state to itself. Each accepting run then takes one σ_{ij} and infinitely many τ_{ik} . The intuition sketched in Figure A.13 also clarifies the structure of ω -regular expressions. Recall that these expressions take the form $(+ i : i \in \mathbb{N} \wedge 0 \leq i < n : \alpha_i \cdot \beta_i^\omega)$ for some $n \in \mathbb{N} \setminus \{0\}$ and α_i and β_i regular expressions such that $\varepsilon \notin \mathcal{L}(\beta_i)$. Any α_i in such an ω -regular expression defines the language of all paths σ_{ij} from an initial state to some final state; any β_i defines the language of all cycles τ_{ik} from that final state to itself. Note that it is crucial to take into account *all* cycles, because after completion of a cycle in an accepting run, any cycle may be selected for the next iteration through the final state. (Note that the regular expression β_0 in the above example indeed captures precisely all cycles through the final state of the considered automaton.) The language of an NBA, and hence the ω -regular expression describing the entire ω -language of the NBA, can then be obtained by taking the union of all possibilities. Note that an NBA may have an ε -labeled cycle from a final state to itself. Such a cycle allows runs that pass a final state infinitely often with a corresponding finite word. Such a word is not accepted by the NBA, since the language of an NBA, by definition, only contains infinite words.

The notions of determinism and completeness for Büchi automata are inherited from the definitions for finite automata.

An NBA is **deterministic** (and referred to as a DBA) if the corresponding NFA is deterministic; it is **complete** if the NFA is complete.

Example A.17 (NBA). Recall Example A.16. The automaton of Figure A.1 is deterministic. Now, let us consider the complement language: $\overline{\mathcal{L}^\omega(B)} = \Sigma^\omega \setminus \mathcal{L}^\omega(B)$. How do we describe its elements? Obviously, we exclude from $\overline{\mathcal{L}^\omega(B)}$ all infinite words that consist solely of b -s and/or c -s; in other words, all words in $\mathcal{L}^\omega(B)$ have at least one a . However, from these infinite words with at least one a , we must exclude the ones that have a b at some point after every occurrence of an a (as required by $\mathcal{L}^\omega(B)$). What is left are infinite words that eventually have an a occurrence that will never be followed again by any b occurrences; so, all infinite words of $\overline{\mathcal{L}^\omega(B)}$ must have an infinite ‘tail’ of only a -s and/or c -s, although they may start out with an initial finite segment possibly containing some b -s. An NBA for $\overline{\mathcal{L}^\omega(B)}$ is in fact depicted in Figure A.3(a). Note that this automaton is nondeterministic and incomplete. Figure A.3(b)

shows its completion.

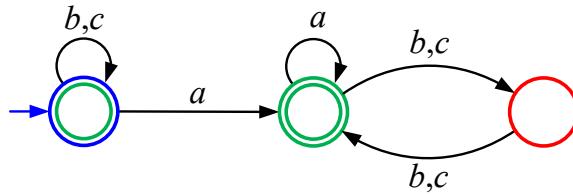


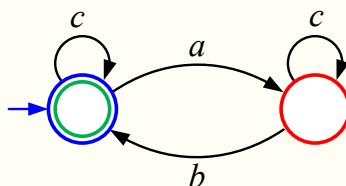
Figure A.14: Another deterministic Büchi automaton.

Example A.18 (DBA). As a last example, we construct a Büchi automaton for the ω -language described as follows: “Between any two a -s, the total number of b -s and/or c -s is even (including 0).” Clearly, a -s need not occur at all, which means that we get the sublanguage of arbitrary infinite words of b -s and c -s, resulting in a state that is both initial and final with a self-loop labeled $\{b, c\}$. If an a occurs in an infinite word, it is okay if the (infinite) remainder of the word consists of only a -s. However, if we include a b or c after an a , we must make sure to enforce an even number of them. Taking all this in consideration, the Büchi automaton of Figure A.14 suggests itself. It is deterministic but incomplete. At a first glance, the automaton might seem to have one fallacy: It is only possible to end a word with an *even* number of b -s and/or c -s; we cannot have an *odd* number. However, since any tail of an infinite string is infinite, it does not make sense to talk about even and odd lengths, which means that the automaton of Figure A.14 is correct.

Exercise A.17 (ω -Regular expressions). Give ω -regular expressions for the ω -languages discussed in Examples A.17 and A.18.

See answer.

Exercise A.18 (Büchi automata). Consider the following Büchi automaton over alphabet $\Sigma = \{a, b, c\}$.



1. Describe in your own words the ω -language that the Büchi automaton accepts. Give an ω -regular expression for this language.
2. Is the Büchi automaton deterministic? Motivate your answer. If not, give a deterministic Büchi automaton accepting the same ω -language.
3. Is the Büchi automaton complete? Motivate your answer. If not, give a complete Büchi automaton accepting the same ω -language.

4. Assume we change the Büchi automaton by turning the non-final state into a final state. Does this change the language accepted by the automaton? Motivate your answer. If the language has changed, give an ω -regular expression for the new language.

See answer.

Exercise A.19 (ω -regular expressions, Büchi automata). Let $\Sigma = \{a, b\}$. Consider the ω -language described as follows:

“All strings that start with an a and contain at least one b .”

1. Give an ω -regular expression specifying this language.
2. Give an NBA that accepts the above language.

See answer.

Exercise A.20 (ω -regular expressions, Büchi automata). Given the alphabet $\Sigma = \{a, b\}$. Consider the following ω -language:

$$L = \{\sigma \in \Sigma^\omega \mid \text{the number of occurrences of } a \text{ in } \sigma \text{ is finite and even}\}$$

1. Give an NBA that accepts ω -language L .
2. Give an ω -regular expression for L .
3. Give an NBA that accepts the complement of ω -language L .

See answer.

Exercise A.21 (Finite automata, Büchi automata). Give example automata B_1 and B_2 such that $\mathcal{L}^\omega(B_1) = \mathcal{L}^\omega(B_2)$ but $\mathcal{L}(B_1) \neq \mathcal{L}(B_2)$. That is, the two automata accept the same ω -language but not the same finite-word language.

See answer.

A.4.3 (Non)determinism in Büchi automata

The attentive reader has noticed that Theorem A.4 (ω -regularity) does not mention DBA as having the same expressivity as NBA. It turns out that *DBA are less expressive than NBA*. This result deviates from the results obtained for finite automata, where DFA and NFA are equally expressive, capturing precisely the class of regular languages.

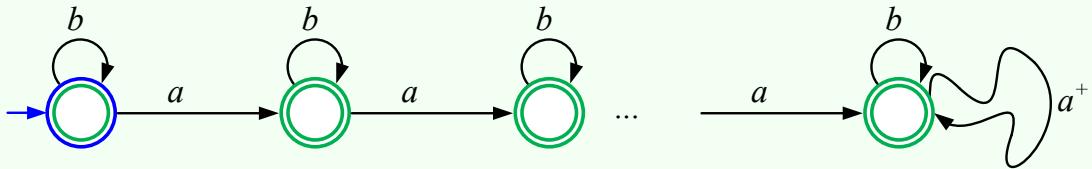
An ω -language is said to be *deterministically Büchi recognizable* if and only if it is recognized by a DBA.

Example A.19 (NBA, DBA, and complements). Let $\Sigma = \{a, b\}$. Consider the following Büchi automata, referred to as B and C , respectively.



It is not difficult to see that the ω -languages accepted by B and C are each others complements. B accepts all words in which infinitely many a -s occur, $\mathcal{L}^\omega(B) = \{\sigma \in \Sigma^\omega \mid a \in \inf(\sigma)\}$; C accepts all words with finitely many a -s, implying that from some point onwards only b -s occur, $\mathcal{L}^\omega(C) = \{\sigma \in \Sigma^\omega \mid a \notin \inf(\sigma)\} = \mathcal{L}^\omega(B)$. Clearly, B is deterministic, whereas C is not.

It can be argued that no DBA exists specifying the same language as NBA C . Assume that such a DBA D exists with $\mathcal{L}^\omega(D) = \mathcal{L}^\omega(C)$. Because of the determinism, it should necessarily be of the following form:



Any finite prefix of a word in $\mathcal{L}^\omega(C)$ should end up in a final state with a b self-loop. There are infinitely many prefixes a^n of n a -s such that $a^n b^\omega$ is a word of $\mathcal{L}^\omega(C)$. Since a DBA has only finitely many states, the pigeonhole principle implies that at some point two prefixes a^n and a^{n+m} (with $m > 0$) must end up in the same state. But this implies that a^ω is an element of $\mathcal{L}^\omega(C)$, contradicting the definition of $\mathcal{L}^\omega(C)$. Hence, DBA D cannot exist.

Since any DBA is also an NBA, this example shows that the class of deterministically-Büchi-recognizable languages is a strict subset of the ω -regular languages. Another immediate consequence of this example is that DBA are not closed under complement.

The class of deterministically-Büchi-recognizable languages is not closed under complement.

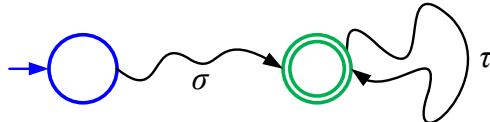
We can precisely characterize the class of deterministically-Büchi-recognizable languages in terms of (finite-word) regular languages. To this end, inspired by the above example, we introduce the notion of the limit language.

Definition A.16 (Limit language). Recall the notion of prefixes of strings, denoted by the binary relation \preceq . Let Σ be some finite alphabet. For any infinite word $\sigma \in \Sigma^\omega$, $\text{pref}(\sigma) \subseteq \Sigma^*$ is the (finite-word) language containing all prefixes of σ : $\text{pref}(\sigma) =$

$\{\tau \in \Sigma^* \mid \tau \preceq \sigma\}$. For any given finite-word language $K \subseteq \Sigma^*$, the *limit language* $\lim(K)$ is the ω -language of all infinite words that have infinitely many prefixes in K : $\lim(K) = \{\sigma \in \Sigma^\omega \mid |\text{pref}(\sigma) \cap K| = \omega\}$.

Proposition A.2 (Limit characterization of deterministically-Büchi-recognizable languages). An ω -language L is deterministically Büchi recognizable if and only if $L = \lim(K)$ for some (finite-word) regular language $K \subseteq \Sigma^*$ with Σ some alphabet.

The following figure illustrates the limit construction.



The figure shows a DFA D with a path labeled σ from the initial state to a final state, and a path labeled τ from that final state to itself. This means that words $\sigma\tau, \sigma\tau^1, \sigma\tau^2, \sigma\tau^3, \dots$ are all elements of the regular language accepted by the DFA. The limit construction states that $\sigma\tau^\omega$ is an element of the limit of this regular language, $\lim(\mathcal{L}(D))$. We already know from the acceptance conditions for Büchi automata that $\sigma\tau^\omega$ is then an element of $\mathcal{L}^\omega(D)$ when considering D as a DBA.

Since for a given word in a language, accepting runs in a *deterministic* automaton are unique, the following is an interesting and immediate consequence of the above reasoning.

Corollary A.1 (Limits of deterministic automata). For any *deterministic* finite automaton D , $\mathcal{L}^\omega(D) = \lim(\mathcal{L}(D))$.

This corollary states that a DBA recognizing a deterministically-Büchi-recognizable ω -language is actually a DFA that specifies a (finite-word) regular language of which the limit is the deterministically-Büchi-recognizable ω -language.

Example A.20 (Limits). Reconsider Example A.19. It is not difficult to see that $\mathcal{L}^\omega(B) = \{\sigma \in \Sigma^\omega \mid a \in \inf(\sigma)\} = \lim(\mathcal{L}(B))$ with $\mathcal{L}(B) = \{\sigma a \mid \sigma \in \Sigma^*\}$. B interpreted as a DFA accepts all finite words ending with an a .

For *nondeterministic* automaton C , it can be shown that no finite-word regular language exists of which the limit language equals $\mathcal{L}^\omega(C)$. Note that automaton C can be interpreted as an NFA, accepting finite-word regular language $\mathcal{L}(C) = \{\sigma b \mid \sigma \in \Sigma^*\}$ containing all finite words ending with a b . So $\mathcal{L}(C)$ seems a good candidate finite-word regular language of which the limit may equal $\mathcal{L}^\omega(C)$. However, the definition of $\mathcal{L}(C)$ implies that $(ab)^n$ for any $n \in \mathbb{N}$ is an element of $\mathcal{L}(C)$. This in turn implies that $(ab)^\omega$, containing infinitely many a -s, is an element of $\lim(\mathcal{L}(C))$. From this, we may conclude that $\lim(\mathcal{L}(C)) \neq \mathcal{L}^\omega(C)$.

So the limit of $\mathcal{L}(C)$ does not give us $\mathcal{L}^\omega(C)$. To show that no finite-word regular language exists of which the limit language equals $\mathcal{L}^\omega(C)$, assume that such a language

K exists with $\mathcal{L}^\omega(C) = \lim(K)$, for some regular language $K \subseteq \Sigma^*$. Since $b^\omega \in \mathcal{L}^\omega(C) = \lim(K)$, there must exist an $n \in \mathbb{N}$ such that $b^n \in K$. Since $b^nab^\omega \in \mathcal{L}^\omega(C)$, there must exist an $m \in \mathbb{N}$ such that $b^nab^m \in K$. Since $b^nab^mab^\omega \in \mathcal{L}^\omega(C)$, there must exist an $l \in \mathbb{N}$ such that $b^nab^mab^l \in K$. It is possible to continue in this way, thus constructing an increasing chain of infinitely many words in K , where each word contains a finite number of a -s with a finite number of b -s between every pair of a -s. As a result, $\lim(K)$ contains an infinite word containing an infinite number of a -s. This clearly contradicts the assumption that $\mathcal{L}^\omega(C) = \lim(K)$, completing the argument that $\mathcal{L}^\omega(C)$ cannot be characterized as a limit.

Let's have a closer look at the chain $b^n, b^nab^m, b^nab^mab^l, \dots$. The word b^n can be executed on automaton C in many ways. However, considering C as an NFA, to accept the word in $\mathcal{L}(C)$, at some point the transition to the final state has to be made. Thus, the left self loop is taken x times for some x , followed by a step to the final state, followed by the right self loop $n - x - 1$ times. Word b^nab^m , however, can only be accepted by taking the left self loop $n + 1 + x$ times for some x , followed by a step from the initial to the final state, followed by the right self loop $m - x - 1$ times. Word $b^nab^mab^l$ must be accepted by taking the left self loop $n + 1 + m + 1 + x$ times, followed by a step from the initial to the final state, followed by the right self loop $l - x - 1$ times. Thus, the state sequences underlying the chain of accepted words $b^n, b^nab^m, b^nab^mab^l, \dots$ do not form a chain themselves (no matter what options for the x -s are selected). This means that the chain of words $b^n, b^nab^m, b^nab^mab^l, \dots$ does not lead to an accepting run in the NBA C . This is caused by the nondeterminism in the initial state of C . This observation illustrates why the requirement that L be deterministically Büchi recognizable in Proposition A.2 is crucial. Note that the concrete example run $(ab)^\omega$ in $\lim(\mathcal{L}(C))$ with its infinitely many prefixes $(ab)^n$ in $\mathcal{L}(C)$ given above form a concrete instance of the generic pattern discussed here.

The above example is an example of a property pattern referred to as ‘eventually forever’, i.e., eventually forever b occurs. This pattern can only be specified in NBA using nondeterminism. Thus, DBA are strictly less expressive than NBA. This is different from the NFA and DFA we've seen earlier. The difference between finite automata and Büchi automata shows the importance of acceptance conditions. The fact that final states in an automaton need to be visited infinitely often fundamentally changes the type of patterns recognized by the automata. The exercises below provide some further insight into these differences.

Proof of Proposition A.2. We now proceed to formally prove the above proposition. To prove the implication from left to right, assume that B is a DBA such that $L = \mathcal{L}^\omega(B)$. It can be shown that $\mathcal{L}^\omega(B) = \lim(\mathcal{L}(B))$, where $\mathcal{L}(B)$ is the finite-word regular language of the DFA B as defined in Definition A.6. First, assume that $\sigma \in \mathcal{L}^\omega(B)$. This means that σ has an accepting run π in which a final state q of B occurs infinitely often. Thus, infinitely many prefixes of this run π end in q . Consequently, infinitely many prefixes of σ are elements of $\mathcal{L}(B)$, which means that $\sigma \in \lim(\mathcal{L}(B))$. Second, assume $\sigma \in \lim(\mathcal{L}(B))$. It follows that $\mathcal{L}(B)$ contains infinitely many (finite) prefixes $\sigma_0, \sigma_1, \dots$ of σ . Since B is deterministic, these prefixes correspond to a *chain* of runs

π_0, π_1, \dots , all ending in some final state of B . Since the number of final states of B is finite, at least one final state must occur infinitely often as the last state in this chain of runs π_0, π_1, \dots . The supremum of this chain is therefore an accepting run for σ in DBA B . Thus, $\sigma \in \mathcal{L}^\omega(B)$. Combining the two cases yields that $\mathcal{L}^\omega(B) = \lim(\mathcal{L}(B))$.

To prove the implication from right to left, assume that $L = \lim(\mathcal{L}(A))$ for some DFA A . Using similar arguments as before, it can be shown that $\mathcal{L}^\omega(A) = \lim(\mathcal{L}(A))$, which means that L is deterministically Büchi recognizable. \square

Exercise A.22 (NB/FA, DB/FA, and limits). Reconsider the Büchi automata B and C of Example A.19.

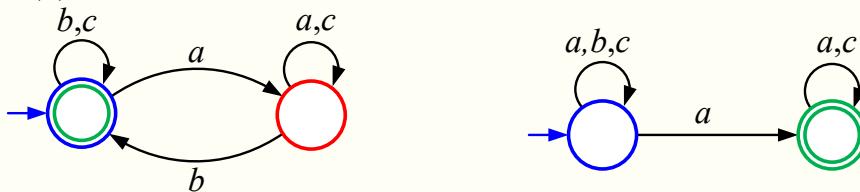


B accepts all words with infinitely many a -s; C accepts all infinite words with finitely many a -s. Consider the regular expression $\alpha = (a + b)^*b$.

1. Give ω -regular expressions for the languages accepted by B and C .
2. Consider B and C as finite automata. Give regular expressions accepting the same languages as these automata. Is $\mathcal{L}(C) = \overline{\mathcal{L}(B)}$? Motivate your answer.
3. Is $\mathcal{L}^\omega((a + b)^*b^\omega) = \lim(\mathcal{L}(\alpha))$? Motivate your answer.
4. Give an NFA (without ε moves) for α with as few states and transitions as possible. What is the ω -regular language accepted by this automaton when considered as an NBA? Give an ω -regular expression for this language.

See answer.

Exercise A.23 (NB/FA, DB/FA, and limits). Recall the automata of Figures A.1 and A.3(a), considered in this exercise as Büchi automata B and C , respectively.



In Examples A.16 and A.17, it was argued that the ω -languages accepted by B and C are each others complements: $\mathcal{L}^\omega(C) = \overline{\mathcal{L}^\omega(B)}$. B accepts all words in which every a is at some point followed by a b (a request-response pattern); C accepts all words that end with a(n infinite) tail consisting of an a without any following b .

1. Give ω -regular expressions for the two languages.

2. Consider B and C as finite automata. Give regular expressions accepting the same languages as these automata. Is $\mathcal{L}(C) = \overline{\mathcal{L}(B)}$? Motivate your answer.
3. Show that $\mathcal{L}^\omega(C) \neq \lim(\mathcal{L}(C))$.
4. (Above exam level) Show that $\mathcal{L}^\omega(C)$ is not deterministically Büchi recognizable (thus showing once more that DBA are not closed under complement).

See answer.

A.4.4 Expressiveness and closure properties

In the previous subsection, we've seen that there is an expressive difference between NBA and DBA. Nondeterminism increases expressiveness in the context of Büchi automata. In this subsection, we further investigate expressiveness aspects by looking at closure properties of ω -regular languages and, briefly, at non-regularity of ω -languages.

Proposition A.3 (Closure properties of ω -regular languages). The class of ω -regular languages is closed under union, intersection, and complement.

The fact that the class of ω -regular languages is closed under union follows directly from its definition, Definition A.14. Closure under complement can be shown through a complement operation on NBA. This complement on NBA is much more involved than the complement on complete DFA given earlier, because of the nondeterminism and the different acceptance conditions. It is omitted here. The interested reader is referred to e.g. [24]. So, where the class of deterministically-recognizable-Büchi automata is not closed under complement, ω -regular languages are. Closure under intersection now follows from the previous two closure properties by observing that $L_0 \cap L_1 = \overline{\bar{L}_0 \cup \bar{L}_1}$ for any languages L_0 and L_1 .

Exercise A.24 (Complement of Büchi automata?). Reconsider the complement for DFA given in Definition A.9. Give an example of a complete DFA A with complement \bar{A} such that $\mathcal{L}^\omega(A) \neq \overline{\mathcal{L}^\omega(\bar{A})}$. Motivate your answer. (That is, the complement construct for DFA does not give the complement language when interpreting the automaton as a Büchi automaton.)

See answer.

So in line with the results for finite-word regular languages, and despite the negative result for DBA, intersection and complement of ω -languages do not yet yield non- ω -regular languages. But as may be expected, there are ω -languages that are not regular.

Exercise A.25 (Non- ω -regular language). Give an ω -language that is not regular. Use the pigeonhole principle to motivate your answer.

See answer.

A.4.5 Conversions

Following Theorem A.4 (ω -Regular languages), we proceed to provide conversions between ω -regular expressions and NBA.

Algorithm A.7 (From ω -regular expression to NBA($-\varepsilon$)). The conversion from expression to NBA- ε follows Definition A.14 (ω -Regular expressions) and the conversion from regular expressions to NFA- ε explained in Algorithm A.2.

The constructions to take the union and concatenation of two automata given in Algorithm A.2 provide NBA- ε for the union and concatenation operations. Note that the construction of an NBA- ε for concatenation uses an ordinary finite automaton as a building block, to represent the left operand of the concatenation.

An NBA- ε for the infinite concatenation of a (finite-word) language represented by an NFA- ε follows the construction of an automaton for the Kleene iteration of regular expressions, introducing an extra initial and final state. An alternative option, not introducing an extra state, is to add ε moves from each final state to each initial state.

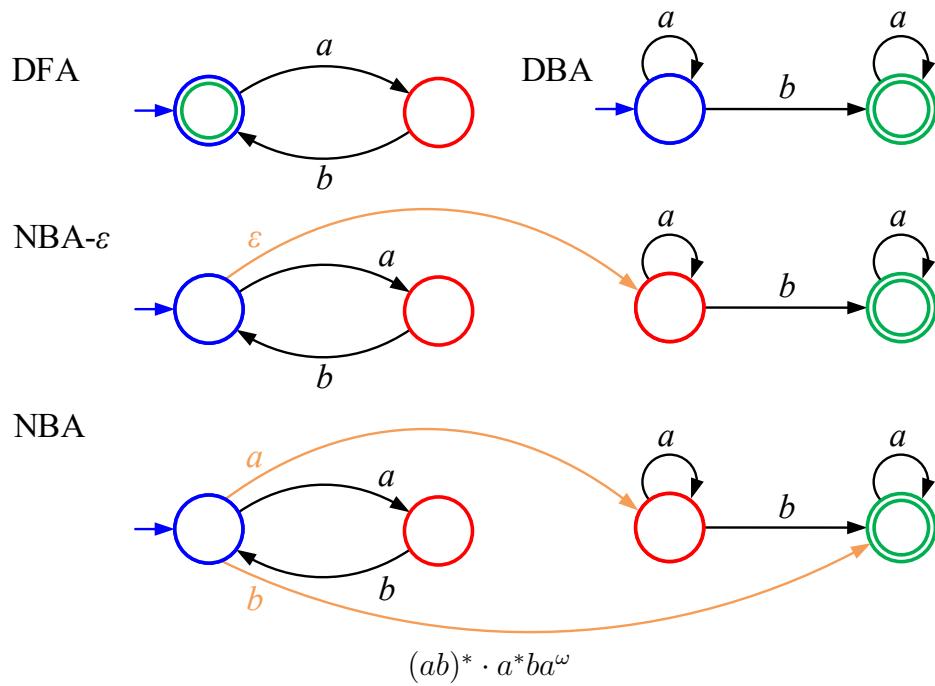
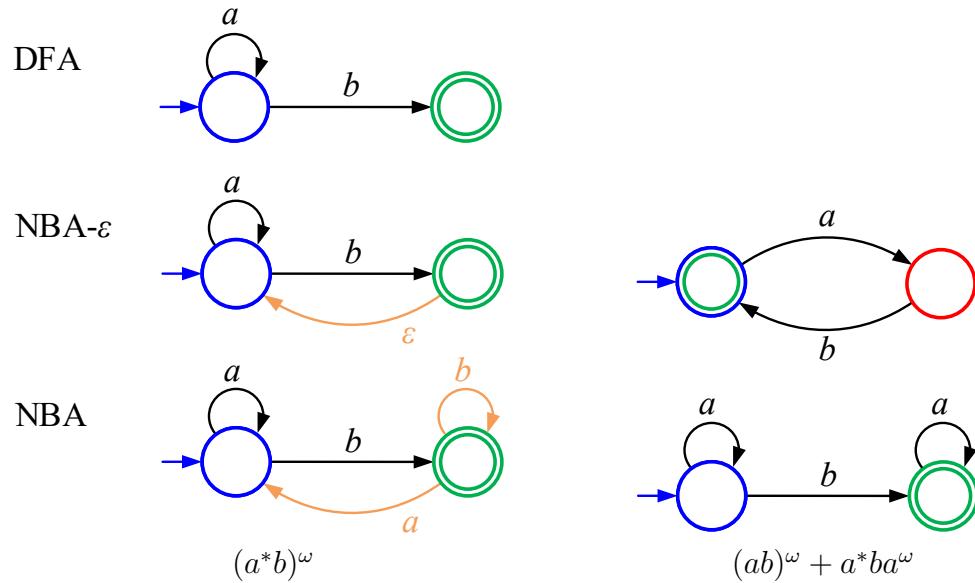
The conversion from an NBA- ε to an NBA goes through the ε elimination already introduced for NFA- ε in Algorithm A.4.

Example A.21 (From ω -regular expression to NBA($-\varepsilon$)). The conversion from ω -regular expression to NBA is illustrated in Figure A.15. For the concatenation and the infinite concatenation, the figure shows the DFA for the finite-word languages and the DBA that are at the basis of the conversion, the step towards an NBA- ε , and the step towards the NBA. The example for the infinite concatenation follows the alternative construction given in the conversion algorithm. The conversion for the union trivially combines the NBA for the two operands into one.

Exercise A.26 (From ω -regular expression to NBA). For each of the following items, create an NBA for the ω -regular expression at hand following Algorithm A.7. Start from finite automata for any finite-word subexpressions that are as compact as possible. First, create an NBA- ε ; then, eliminate ε moves to arrive at an NBA.

1. Create an NBA for the ω -regular expression given in Example A.16.
2. Create an NBA for the ω -regular expression given in Example A.15.
3. Create an NBA for ω -regular expression $\alpha\beta\gamma^\omega$ with $\alpha = a(ba)^*$, $\beta = a^*b$, and $\gamma = a$.
4. Consider the coffee machine of Example A.14. Create an NBA for the ω -regular expression for the continued service of providing proper sales transactions given in Exercise A.16.3.

See answer.

Figure A.15: Conversion from ω -regular expressions to NBA.

It remains to consider the conversion from NBA to ω -regular expressions. This conversion follows the format of ω -regular expressions as they are defined in Definition A.14 (ω -Regular expressions) and the illustration of the acceptance conditions of NBA in Figure A.13.

Algorithm A.8 (From NBA(- ε) to ω -regular expression). Let $B = (Q, \Sigma, \Delta, Q_0, F)$ be an NBA(- ε). Let A_{ij} be the NFA $A_{ij} = (Q, \Sigma, \Delta, \{i\}, \{j\})$. That is, A_{ij} accepts the finite-word regular language $\mathcal{L}(A_{ij})$ of all words going from state i to state j in B . Language $\mathcal{L}^\omega(B)$ can now be specified as $(\cup q_0, q_f : q_0 \in Q_0 \wedge q_f \in F : \mathcal{L}(A_{q_0 q_f}) \cdot \mathcal{L}(A_{q_f q_f})^\omega)$. Using Algorithm A.3 (State elimination), each NFA A_{ij} can be converted into finite-word regular expression α_{ij} that accepts the same language. ω -Regular expression $(+ q_0, q_f : q_0 \in Q_0 \wedge q_f \in F : \alpha_{q_0 q_f} (\alpha_{q_f q_f})^\omega)$ then specifies $\mathcal{L}^\omega(B)$.

Example A.22 (From NBA to ω -regular expression). Let $\Sigma = \{a, b\}$. Consider the DBA B (from Example A.19) in the top left of Figure A.16. The rest of the figure shows how regular expressions $\alpha_{12} = b^*a((a+b)b^*a)^*$ and $\alpha_{22} = \varepsilon + (a+b)(b+a(a+b))^*a$ can be obtained through state elimination. This leads to ω -regular expression $\beta = b^*a((a+b)b^*a)^*(\varepsilon + (a+b)(b+a(a+b))^*a)^\omega$, with $\mathcal{L}^\omega(\beta) = \mathcal{L}^\omega(B)$. Expression β can be simplified, to, e.g., $b^*a((a+b)b^*a)^*((a+b)b^*a)^\omega$ and ultimately to $b^*a((a+b)b^*a)^\omega$.

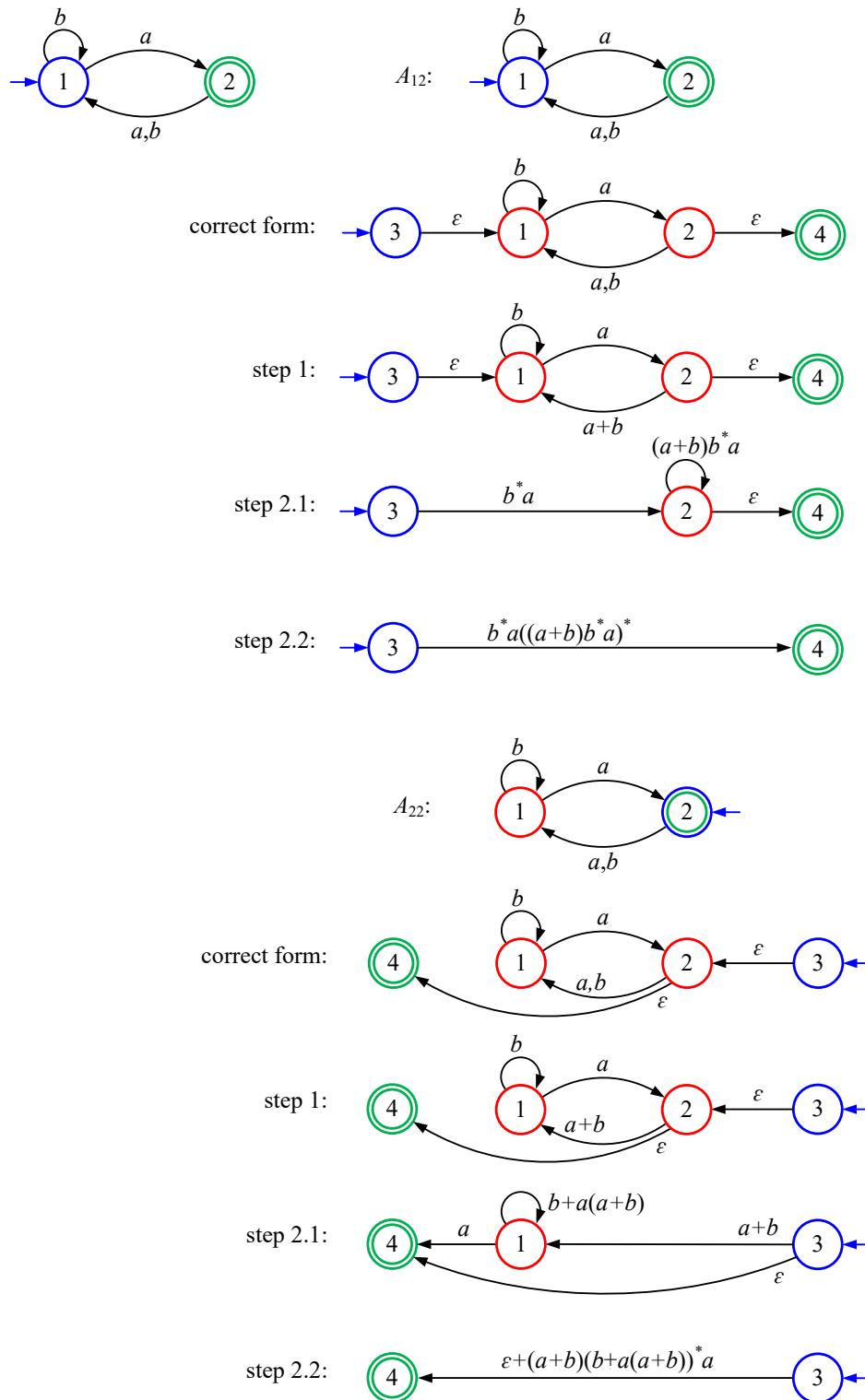
Exercise A.27 (From NBA to ω -regular expression).

1. Consider the DBA of Example A.18. Create an ω -regular expression for this DBA with Algorithm A.8. Compare your answer to the regular expression you gave in Exercise A.17.
2. Consider the three NBA of Figure A.15. Create ω -regular expressions for these NBA with Algorithm A.8. Compare your answers to the regular expressions given in Figure A.15.

See answer.

A.4.6 Decidability issues

We've come to the point that the theory of ω -languages has been developed along the same lines as the theory of finite-word languages earlier. This naturally raises questions like “Given a word, how do we check its membership of a given ω -regular language?”, “Are these two languages equal?”, and “How can we verify the validity of ω -regular properties, i.e., properties specified as an ω -regular language, for a given system?”. Since we are now dealing with infinite words, it may not be obvious that these questions are still decidable. But it turns out that they are. Checking the emptiness of an ω -regular language is fundamental to answering all these questions. Note that we cannot simply execute an infinite word on a Büchi automaton to check membership. Checking whether a given word is an element of a given ω -language can be done by checking whether the intersection of this language and the language containing only the given word is nonempty. Checking language equivalence

Figure A.16: Conversion from NBA to ω -regular expression.

reduces to two inclusion checks, which reduces to emptiness checks of the intersections of one language with the complement of the other and vice versa. And we have already seen in Section A.3 that property checking boils down to an inclusion check, or alternatively to checking emptiness of the intersection of the language of system behaviors and the language of bad behaviors. Being able to decide the emptiness of a language therefore has far-reaching practical implications. Emptiness checking turns out to be decidable for ω -regular languages.

Deciding whether an ω -regular language is empty. An NBA is a graph with finitely many nodes and edges. By a depth-first search of that graph starting from each of the initial states, it can be decided whether any of the final states of the NBA are reachable. From each of these final states, another depth-first search can be started to see whether it lies on a cycle. If any of the final states reachable from one of the initial states lies on a cycle, the language is not empty. Otherwise, it is. As already mentioned, a depth-first search of a graph can be efficiently implemented with a complexity that is linear in the size of the graph. The approach sketched here is a nested depth-first search that can be further refined and optimized. The interested reader is referred to [3] for details.

A.4.7 Checking ω -regular properties

As already explained in Section A.3, the automata-based approach sketched in that section to verify properties is not limited to finite-word regular languages. Algorithm A.6 (Checking regular properties) can be adapted to ω -regular properties, giving Algorithm A.9. Recall that property checking is essentially checking the inclusion of the system behaviors in the property behaviors. Via complement and intersection, this translates to an emptiness check. However, taking the complement of an ω -regular language is quite involved and cannot be done efficiently. Therefore, it is common practice to start property checking for ω -languages from the bad behaviors, instead of from the good (property) behaviors. Algorithm A.9 (Checking ω -regular properties) essentially follows Algorithm A.6 for regular languages, but with two important differences. First, because it is not possible to efficiently convert any NBA to a complement NBA, the first two steps of Algorithm A.6 are collapsed into a single step for ω -regular languages. Instead of specifying an automaton for the property of interest, the algorithm starts with an NBA for the complement language, the language specifying the bad behaviors. Second, similar to Algorithm A.6, Algorithm A.9 computes the intersection of the two languages at hand via a product automaton. But the product construction for NFA does not work for NBA. To compute the intersection of ω -regular languages, Algorithm A.9 uses the notion of a *generalized* NBA (GNBA) with a product defined on those generalized NBA, as further explained below.

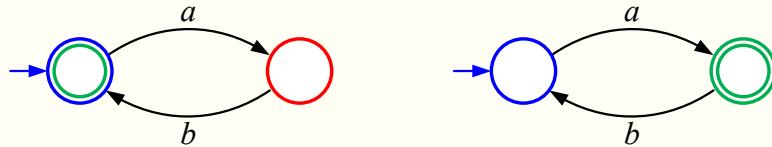
Algorithm A.9 (Checking ω -regular properties). Let P be an ω -regular language specifying the property of interest. Let S be an ω -regular language describing all system behaviors. It can be verified as follows whether $S \models P$:

1. Create an NBA $B_{\bar{P}}$ specifying the bad behaviors \bar{P} ;

2. create an NBA B_S for S ;
3. create the product GNBA $B_S \otimes B_{\bar{P}}$;
4. check for emptiness of $\mathcal{L}^\omega(B_S \otimes B_{\bar{P}})$.

If the language $\mathcal{L}^\omega(B_S \otimes B_{\bar{P}}) = \emptyset$, then $S \models P$, i.e., system S validates property P .

Exercise A.28 (Product automata). Let Σ be alphabet $\{a, b\}$. Consider the following two automata, A and B , respectively:



Create the product $A \times B$, following Definition A.10. Argue that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$, but that $\mathcal{L}^\omega(A \times B) \neq \mathcal{L}^\omega(A) \cap \mathcal{L}^\omega(B)$.

See answer.

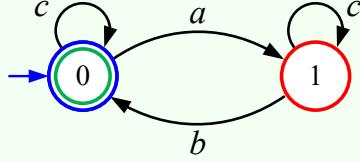
The key reason that the product for NFA does not work for NBA lies in the acceptance conditions. A finite word is an element of the intersection of two finite-word regular languages if it is recognized by the two corresponding automata, i.e., if it ends up in a final state for both the automata. That aligns with the product construction that has the Cartesian product of the two sets of final states of the operand automata as its set of final states. This does not align though with the acceptance conditions of Büchi automata. An infinite word is an element of the intersection of two ω -regular languages if it has an accepting run in both automata, i.e., if it has a run in each of the automata that goes through a final state of those automata infinitely often. But it does not have to be in a final state of the two automata simultaneously each time, which is what the product automaton for finite automata enforces. So we need a product construct that accepts a word if it visits each of the final-state sets of the operand automata infinitely often. This observation leads naturally to the notion of the already mentioned *generalized* Büchi automaton. A generalized NBA is an NBA that has a set of final-state sets, instead of just one such set, each of which needs to be visited infinitely often in an accepting run. As before, GNBA may include ε moves. For simplicity, we do not distinguish notations for the two classes of GNBA with and without ε moves.

Definition A.17 (Generalized NBA (GNBA)). Let Σ be some finite alphabet. Then $B = (Q, \Sigma, \Delta, Q_0, \mathcal{F})$ with Q, Δ , and Q_0 as for an NBA and $\mathcal{F} \subseteq 2^Q$ is a *generalized* NBA. A word $\sigma \in \Sigma^\omega$ is *accepted* by B if and only if there is an infinite run $\pi \in Q^\omega$ such that $\pi(0) \in Q_0$, for all $i \in \mathbb{N}$, $\pi(i) \xrightarrow{\sigma(i)} \pi(i+1)$, and for all $F \in \mathcal{F}$, $\inf(\pi) \cap F \neq \emptyset$. The ω -language of B , denoted $\mathcal{L}^\omega(B)$ as before, is the set of all infinite words accepted by B .

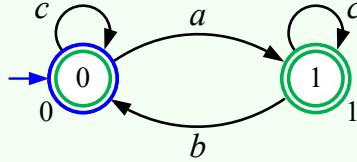
It is clear that any NBA is a GNBA, so GNBA allow to specify at least all ω -regular languages. It turns out though, that no new languages can be specified. That is, the expressiveness of NBA and GNBA is the same.

Proposition A.4 (Expressiveness of GNBA). GNBA allow to specify precisely the class of ω -regular languages.

Example A.23 (GNBA). Consider the following NBA over alphabet $\Sigma = \{a, b, c\}$.

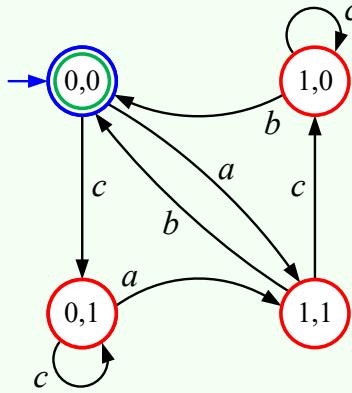


It accepts all words in which every a (if one occurs) is always followed by a b , a so-called request-response pattern. Presume now that we would like to add the requirement that the number of a -s (and hence the number of b -s) is infinite. We cannot do this with an NBA with only two states, since making any of these states final allows infinite tails of only c -s. But we can achieve this with a two-state GNBA:



This figure uses an index next to each final state to indicate of which final-state set it is a member. The shown GNBA thus has two final-state sets, $F_0 = \{0\}$ and $F_1 = \{1\}$, each with a single member. Each of these final-state sets, and hence each of the final states, has to be visited infinitely often. Thus, this GNBA accepts the ω -regular language in which a -s and b -s alternate and occur infinitely often, starting with an a .

Proposition A.4 implies though that it should be possible to specify the same language with an NBA. Any GNBA can be transformed to an NBA by taking a copy of the automaton for each final-state set, connecting transitions from states in the final-state set for a given copy to the corresponding result states of the next copy (where the last copy is connected to the first one again), and taking the final-state set corresponding to the first copy as the final states of the resulting NBA. For this particular example, this yields the following result, where the states contain a pair of indices corresponding to the state identifier and the final-state-set identifier.



Any accepting run of this NBA necessarily goes through both the final-state sets of

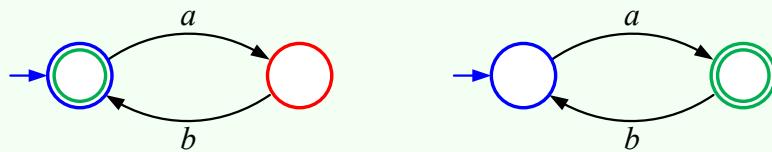
each of the two copies of the automaton infinitely often. It is not difficult to verify that the ω -language accepted by this NBA is the desired one.

Note that the construct sketched in the above example to convert a GNBA to an NBA enforces an order between the final-state sets of the original GNBA that is not present in the definition of the language accepted by a GNBA. But because all the final-state sets need to be visited infinitely often, they are also visited infinitely often in any particular order.

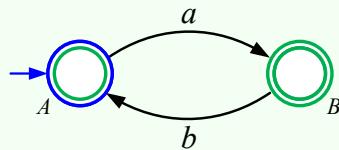
With GNBA, it is straightforward to define a product automaton that accepts the intersection of ω -regular languages. The only difference with the product of NFA, Definition A.10, is how the sets of final states are defined. Intuitively, the product takes the union of the sets of final-state sets of the two automata. But since the product automaton has pairs of states of the operand automata as its states, the final-state sets are combined with the states of the other automaton, where it does not matter in which state the other automaton is. For convenience, we define the product for GNBA without ε moves only. ε Moves can be eliminated with the ε elimination introduced before (straightforwardly generalized to cope with multiple final-state sets).

Definition A.18 (Product GNBA). Let $B_1 = (Q_1, \Sigma, \Delta_1, Q_{01}, \mathcal{F}_1)$ and $B_2 = (Q_2, \Sigma, \Delta_2, Q_{02}, \mathcal{F}_2)$ be two GNBA without ε moves and with the same alphabet Σ . The product GNBA $B_1 \otimes B_2 = (Q_1 \times Q_2, \Sigma, \Delta_\times, Q_{01} \times Q_{02}, \mathcal{F}_{1\times} \cup \mathcal{F}_{2\times})$ with, for all $(q_1, q_2) \in Q_1 \times Q_2$ and $a \in \Sigma$, $\Delta_\times((q_1, q_2), a) = \Delta(q_1, a) \times \Delta(q_2, a)$, and with $\mathcal{F}_{1\times} = \{F_1 \times Q_2 \mid F_1 \in \mathcal{F}_1\}$ and $\mathcal{F}_{2\times} = \{Q_1 \times F_2 \mid F_2 \in \mathcal{F}_2\}$. The ω -language of the product GNBA is the intersection of the languages of the original two automata: $\mathcal{L}^\omega(B_1 \otimes B_2) = \mathcal{L}^\omega(B_1) \cap \mathcal{L}^\omega(B_2)$.

Example A.24 (Product GNBA). Consider again the following two automata, A and B , given in Exercise A.28:



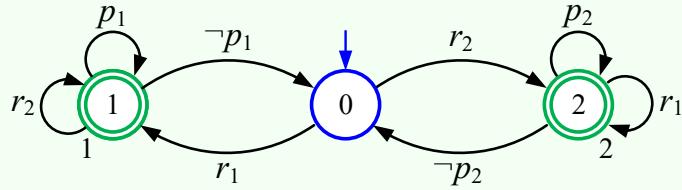
Interpret these automata as GNBA, each with a single final-state set. The product $A \otimes B$, following Definition A.18, is then as follows:



The product GNBA has two states (unreachable states being omitted) and two final-state sets, indexed with A and B to indicate that they originate from automata A and B , respectively. It is clear that $\mathcal{L}^\omega(A \otimes B) = \mathcal{L}^\omega(A) \cap \mathcal{L}^\omega(B)$. In fact, all three automata, A , B , and $A \otimes B$ accept the same ω -language.

Example A.25 (Checking ω -regular properties). Consider a system of two continuously active software processes P_1 and P_2 executing on a shared processor. The processes can request access to the processor through a request action r_i , with $i \in \{1, 2\}$. If granted access, processes can run, indicated with a p_i (process) action. After processing some time, a running process will stop processing, either on its own request or enforced by the system, indicated with a $\neg p_i$ action. If a process requests access to the processor while the other process is running, then the request is not granted (no pre-emption). It may be assumed that no two actions can occur exactly at the same time. That is, two actions will always occur in some order. The system enforces some fair sharing of the processor.

The behavior of this system can be specified naturally as an ω -language. The unbounded behaviors of the system of continuously running processes are abstracted through infinite words. The system can be modeled by the following GNBA S :



This GNBA has two final-state sets, $F_1 = \{1\}$ and $F_2 = \{2\}$, implying that each of the states 1 and 2 has to be visited infinitely often for a word to be accepted. This aligns with the mentioned fair sharing policy.

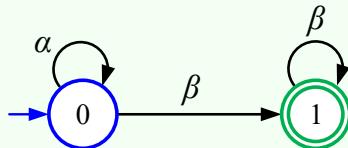
We now want to verify the property P that process P_1 gets the processor infinitely often during any infinite run of the system S . This is the case if P_1 executes an r_1 action infinitely often.^a Let $\alpha = r_1 + p_1 + \neg p_1 + r_2 + p_2 + \neg p_2$. The desired behaviors, property P , can then be specified by:

$$(\alpha^* r_1)^\omega$$

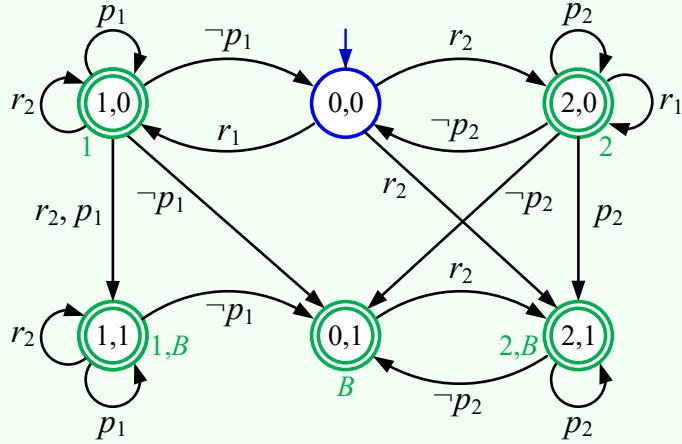
The property-checking algorithm, Algorithm A.9, requires an NBA of the bad behaviors though, not of the expected behaviors. Assuming $\beta = p_1 + \neg p_1 + r_2 + p_2 + \neg p_2$ (i.e., α minus the r_1 option), the bad behaviors can be specified as:

$$\alpha^* \beta^\omega$$

That is, a bad behavior is characterized by an infinite tail without r_1 action. In the form of an NBA B , this yields the following:



At this point, Algorithm A.9 asks for the product automaton $S \otimes B$:



This product has *three* final-state sets $F_{1\times} = \{(1, 0), (1, 1)\}$, $F_{2\times} = \{(2, 0), (2, 1)\}$, and $F_{\times B} = \{(0, 1), (1, 1), (2, 1)\}$. It is not difficult to verify that no run of $S \otimes B$ visits all these three state sets infinitely often. Once the step towards any of the $(i, 1)$ states is made, it is no longer possible to return to any of the $(i, 0)$ states. And within the $(i, 1)$ states, it is impossible to visit both $F_{1\times}$ and $F_{2\times}$ states infinitely often. Hence, the language of $S \otimes B$ is empty. Or in other words, the intersection of the system behaviors and the bad behaviors is empty. Hence, our system satisfies the property P .

^aNote that executing r_1 is the only way to enter state 1, which needs to be visited infinitely often; any r_1 sequence in state 2 is moreover necessarily finite because state 2 needs to be left again each time it is entered in order to visit state 1 again.

Exercise A.29 (Checking ω -regular properties). Consider the system S and property P as specified in Example A.25.

1. Consider a seemingly simpler specification of the desired behavior for process P_1 , stating that P_1 always executes p_1 infinitely often. Verify whether system S satisfies this property following the steps of Algorithm A.9. If not, give a word accepted by S that is a counter example.
2. Specify the property that both P_1 and P_2 always receive access to the shared processor infinitely often as an ω -regular language. Verify whether or not system S satisfies this property. If not, give a word accepted by S that is a counter example.
3. Consider the automaton for S given in Example A.25 as an NBA with a single final-state set $F = \{1, 2\}$. That is, the system does no longer enforce fair sharing, but only guarantees to provide service. Verify whether or not this alternative model of S satisfies the property that both P_1 and P_2 receive access to the shared processor infinitely often.
4. Assume now that P_1 and P_2 can pre-empt each other, under certain conditions:

- P_1 has priority over P_2 and can pre-empt the processing of P_2 , even if no p_2 action was performed yet, with an r_1 action and continue with p_1 actions (or a $\neg p_1$);
- P_2 can pre-empt the processing of P_1 with an r_2 action, but only after at least one p_1 action was performed or if two r_2 -s are issued in a row;
- if P_2 pre-empts P_1 in line with the previous condition, it always performs at least one p_2 action, where the first of these p_2 actions cannot be pre-empted by P_1 .

Specify a new NBA system model for this system, assuming that all states of the NBA are final. That is, the proper behavior cannot be enforced through the final states of the system model; all specified behaviors are part of the accepted language. The system should satisfy the property that two subsequent r_2 actions are always immediately followed by a p_2 action. Specify the unacceptable behaviors as an NBA and verify whether your model indeed satisfies the mentioned property.

See answer.

Exercise A.30 (Checking ω -regular properties). Consider the coffee machine of Example A.14 and the ω -regular property of Exercise A.16.3 specifying its continuous proper service. Verify whether or not the coffee machine satisfies this property.

See answer.

A.5 Linear-Time Temporal Logic

In the earlier sections, we have seen how languages and automata form the basis of a means to specify properties and system behaviors, and to verify whether the specified behavior satisfies a property. Properties can be specified using automata or regular expressions. However, this is not always convenient. Many property specifications refer to only a limited number of actions of the system at hand. But when specifying properties in the form of automata or regular expressions, it is needed to explicitly include also the allowed occurrences of actions that are of no interest. This may lead to unnecessarily complex specifications. This problem gets worse for bigger systems with larger behavioral models. *Logics* provide an alternative means to specify languages that are particularly tuned to the specification of properties. In this section, we study one specific logic, namely Linear-time Temporal Logic, or LTL.

Traditional *propositional* logic provides a means to reason about actions or events at a given point in time. The idea of *temporal* logic is to provide a means to reason about system behavior as it evolves over time. Drawing an analogy with circuit design, propositional logic allows reasoning about combinational circuits, i.e., systems without memory, whereas temporal logic provides a counterpart for sequential circuits, i.e., systems with memory.

A few words about time. As the name suggests, temporal logic has to do with time. Interestingly, despite the name, many temporal logics abstract from precise timing. LTL is an example of such a logic. The way we model time in system design depends on our needs. Many computers and digital systems are driven by some underlying synchronous hardware clock. Such a clock discretizes the underlying continuous physical behavior. It is then natural to consider the behavior of systems only at certain points in time. This yields a *discrete view on time*. The computational model of formal languages adheres to this view, considering only actions and states of systems at specific points in time, while abstracting from precise timing. And although a discrete-time model fits naturally with computers and digital systems, also for asynchronous systems (i.e., systems without global clock) or hybrid systems (i.e., systems with both discrete and continuous properties), it often suffices to only analyze the system at discrete points in time (with or without considering precise timing).

A.5.1 Propositional logic

A logic is a formal framework for constructing statements that can be either true or false. In our context, logic is used to formalize statements about systems, properties that systems should satisfy. A classical example of a logic is so-called propositional logic.

Propositional logic. Propositional logic builds on a set of basic statements, or atomic propositions, Π . From these atomic propositions and the basic constant `true`, more complex statements, called *formulae*, are built using Boolean operators^a:

$$\phi ::= \text{true} \mid \Pi \mid \phi_1 \wedge \phi_2 \mid \neg\phi$$

This rule specifies that a propositional-logic formula ϕ is either the constant `true`, an atomic proposition from Π , a conjunct (and, \wedge) of two formulae ϕ_1 and ϕ_2 , or the negation (\neg) of a(nother) formula ϕ . Note that the constant `false` is not mentioned, and neither are well-known logic operators like ‘or’ (\vee), ‘implies’ (\rightarrow), etc. The reason is that these formulae can be specified in terms of the basic syntax given by the above rule (with ϕ_1 and ϕ_2 propositional formulae):

$$\begin{array}{lll} \text{false} & \stackrel{\text{def}}{=} & \neg\text{true} \\ \phi_1 \vee \phi_2 & \stackrel{\text{def}}{=} & \neg(\neg\phi_1 \wedge \neg\phi_2) & \text{‘or’} \\ \phi_1 \oplus \phi_2 & \stackrel{\text{def}}{=} & (\neg\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \neg\phi_2) & \text{‘exclusive or’} \\ \phi_1 \rightarrow \phi_2 & \stackrel{\text{def}}{=} & \neg\phi_1 \vee \phi_2 & \text{‘implies’, or, ‘if , then ’} \\ \phi_1 \leftrightarrow \phi_2 & \stackrel{\text{def}}{=} & \phi_1 \rightarrow \phi_2 \wedge \phi_2 \rightarrow \phi_1 & \text{‘if and only if’} \end{array}$$

^aThe format used here to specify propositional logic is (a liberally used version of) the well-known Backus-Naur Form (BNF) that specifies words in a language (!) through grammar rules giving all alternative formats for such words. The language at hand is the language of propositional-logic formulae.

Semantics of propositional-logic formulae. An *evaluation* assigns a truth value true or false to each of the atomic propositions in Π . An evaluation can be represented by a set of atomic propositions $\mu \subseteq \Pi$, specifying those atomic propositions that are considered true (implying that the atomic propositions in $\Pi \setminus \mu$ are considered false). The following defines when an evaluation μ satisfies propositional-logic formula ϕ , denoted $\mu \models \phi$.

$$\begin{aligned}\mu &\models \text{true} \\ \mu &\models p \quad \text{iff}^a \quad p \in \mu \quad (p \in \Pi) \\ \mu &\models \phi_1 \wedge \phi_2 \quad \text{iff} \quad \mu \models \phi_1 \text{ and } \mu \models \phi_2 \\ \mu &\models \neg\phi \quad \text{iff} \quad \text{not } \mu \models \phi\end{aligned}$$

^a‘iff’ is an abbreviation of ‘if and only if’

As mentioned, we would like to use logic to specify properties of systems. System behaviors are specified as words over some alphabet. So we would like to specify properties in terms of the symbols in such an alphabet (and words built from those symbols).

Propositions over an alphabet of symbols. Let Σ be some alphabet of symbols. Let Π be the set of atomic propositions of interest, with $\Pi \subseteq 2^\Sigma$. That is, atomic propositions are sets of symbols of the alphabet of interest. The intended interpretation is that an atomic proposition is true for a given symbol in the alphabet if and only if that symbol is an element of the set specified by the proposition. That is, we only consider evaluations μ of the atomic propositions of a specific form: for any symbol $s \in \Sigma$ and atomic proposition $p \in \Pi$, $p \in \mu$ if and only if $s \in p$. We then write $s \models p$.

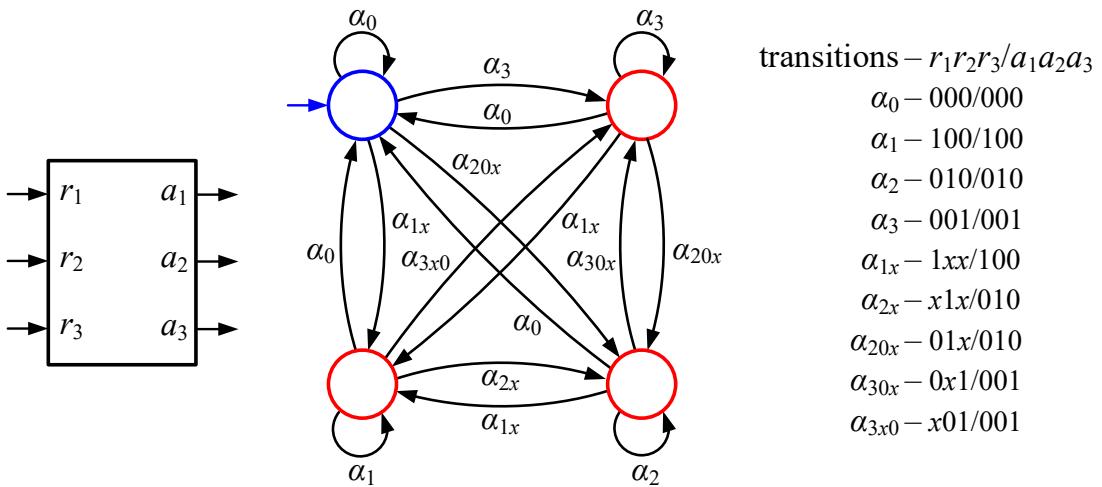


Figure A.17: An arbiter circuit with its Mealy-machine model.

Example A.26 (An arbiter). Let’s assume we want to develop an arbiter circuit that can arbitrate between three different processors competing for a shared resource, such as for example a memory bus. Such a 3-way arbiter circuit has three request inputs

r_1, r_2 , and r_3 , and three acknowledgment outputs a_1, a_2 , and a_3 . Based upon incoming requests (high r_i signals), the arbiter grants requests (at most one at a time) by making the corresponding acknowledgment output high. Figure A.17 shows a Mealy-type state diagram specifying the arbiter. Such a state diagram can be used to synthesize circuit implementations. The Mealy diagram has an initial state marked with a small incoming arrow, following the convention also used for automata. The transitions are of the form $r_1r_2r_3/a_1a_2a_3$, specifying the output pattern $a_1a_2a_3$ generated when input pattern $r_1r_2r_3$ is received in the given state. An x denotes a ‘don’t care’, meaning that the corresponding signal value can be either high or low. A transition with one ‘don’t care’ among the inputs is thus a shorthand for two concrete transitions; for example, $01x/010$ means either $010/010$ or $011/010$. A transition with two don’t cares is a shorthand for four transitions. Note that each state has outgoing transitions for all possible input combinations, in line with the fact that a digital circuit cannot prevent a specific value to occur on any of its inputs.

We would like to investigate the validity of certain correctness properties that we expect from the arbiter. One example of such a property is that no two requests are granted at the same time (*mutual exclusion*). Another example of a desirable property is that no acknowledgment is given if no corresponding request exists (*no waste*). And another example is that every processor eventually gets its turn (*fairness*).

To specify these properties and to verify their satisfaction, we need to first define a proper set of symbols Σ and specify the behavior of the arbiter as an ω -regular language. We choose to define the symbols from all the positive r_i and a_i input and output signals that may occur in a single transition of the arbiter Mealy model: that is, Σ is the set $\{n, r_1a_1, r_1r_2a_1, r_1r_3a_1, r_1r_2r_3a_1, r_2a_2, r_1r_2a_2, r_2r_3a_2, r_1r_2r_3a_2, r_3a_3, r_1r_3a_3, r_2r_3a_3, r_1r_2r_3a_3\}$, where n captures the case that all r_i and a_i input and output signals are low. With these symbols, we can turn the Mealy model of the arbiter into a DBA, as shown in Figure A.18. Since any input/output sequence of a digital circuit that aligns with its Mealy-model specification is a possible behavior of the circuit, all states in the DBA are final states. The DBA specifies the behavior of the arbiter circuit as an ω -language.

To formalize the above mentioned properties, we need to define appropriate atomic propositions Π . Let req_i be equal to $\{s \in \Sigma \mid r_i \in s\}$, i.e., all symbols in Σ that contain r_i . Observe that we use the ‘element of’ notation on strings to denote that a symbol is an element of a string (see the earlier intermezzo on strings). Let $grnt_i$ be equal to $\{s \in \Sigma \mid a_i \in s\}$, i.e., all symbols with a_i . The mutual-exclusion property mentioned earlier can now be formalized as a propositional-logic formula: $\neg(grnt_1 \wedge grnt_2) \wedge \neg(grnt_1 \wedge grnt_3) \wedge \neg(grnt_2 \wedge grnt_3)$. The no-waste property can be stated as $grnt_1 \rightarrow req_1 \wedge grnt_2 \rightarrow req_2 \wedge grnt_3 \rightarrow req_3$.

The mutual-exclusion and no-waste properties refer to the properties of the arbiter for specific moments in time. The fairness property, however, refers to the behavior of the arbiter *over* time. If multiple requests arrive at the same time, the arbiter can only grant one immediately; the other requests must be deferred. Properties that refer to the behavior of a system over time cannot be expressed using propositional logic only.

Thus, we need to extend the logic with operators that allow us to express temporal properties. The class of logics that has such operators is the class of temporal logics, of which LTL is a prime example. The property that every processor *eventually* gets its turn is a *liveness property*. The mutual-exclusion property is an example of a *safety property*. Informally, liveness properties express that something good will eventually happen; safety properties express that nothing bad will ever happen. Safety properties are often specified in the form of *invariants*, i.e., properties that should *always* hold. Note that also the specification of safety properties for systems needs temporal operators. A propositional-logic formula refers to one specific moment in time. A temporal operator is needed to express the fact such a property needs to hold at any moment in time, i.e., *always*. The already mentioned mutual-exclusion and no-waste properties are examples of invariants. The above formalizations only capture the propositional parts of these invariants.

The fairness property can now be made more precise by observing that each processor *always eventually* either gets the resource granted or stops requesting it. The propositional part of the fairness property can be stated as $grnt_i \vee \neg req_i$, for each of the processors i . Above, fairness was more informally phrased as ‘every processor eventually gets its turn’. It was left implicit that this only needs to hold when a processor is continuously asking. The more precise phrasing makes this aspect explicit. It also makes explicit, through the use of ‘always’, that fairness covers all requests. Without this, it would be possible to read the fairness property as a requirement that only needs to be satisfied once. Thus, fairness is a liveness property that is invariant.

The next subsection extends propositional logic with, among others, ‘always’ and ‘eventually’ operators.

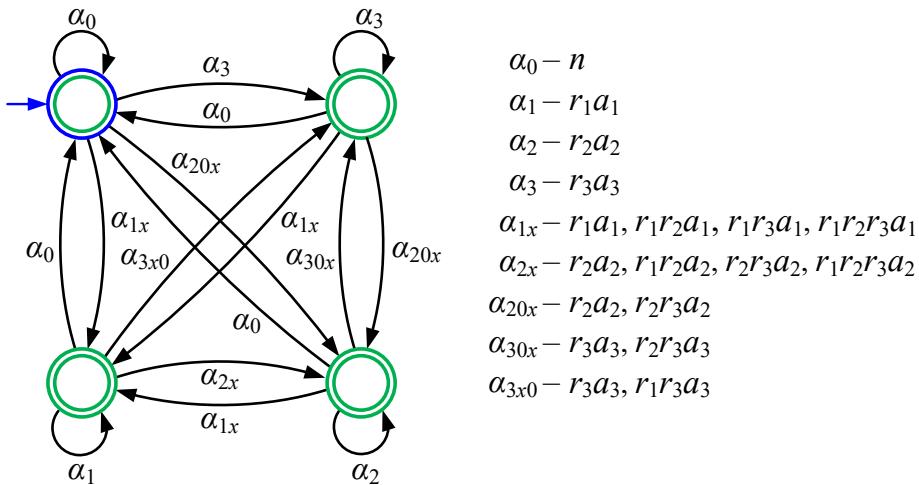


Figure A.18: A DBA for the arbiter.

Exercise A.31 (Propositional logic). Consider the arbiter of Example A.26. Specify the property that, if at least one of the processors requests access, then at least one of them gets the resource (*service provision*). That is, there is no unnecessary idling. See answer.

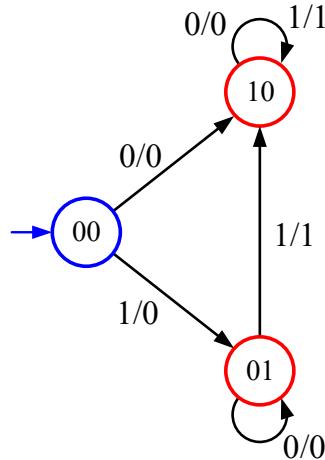


Figure A.19: A Mealy machine.

Exercise A.32 (From Mealy model to DBA). Consider the Mealy-type finite-state machine of Figure A.19. Assume the machine is implemented using two flipflops p and q such that the shown state encoding corresponds to qp , e.g., state 10 corresponds to $qp = 10$. The machine has one input i and one output o . Assume that we want to verify (temporal) properties expressed in terms of the values of Boolean variables p , q , i , and o , e.g., the property that an occurrence of o always results in the q bit being set.

1. Give a DBA representing the Mealy state machine that can be used for this purpose.
2. Define a propositional formula that captures the property that the occurrence of o results in the q bit being set.

See answer.

Exercise A.33 (From Mealy model to DBA - a controller). Consider the Mealy-type finite-state machine of Figure A.20, modeling a controller. Assume the machine is implemented using three flipflops p , q , and r , in such a way that each state n is coded as the binary number rqp with decimal value n , e.g., the initial state 0 corresponds to $rqp = 000$ and state 6 corresponds to $rqp = 110$. We would like to verify a number of properties of this machine. One property is the following:

The initial state is not reachable from any of the other states.

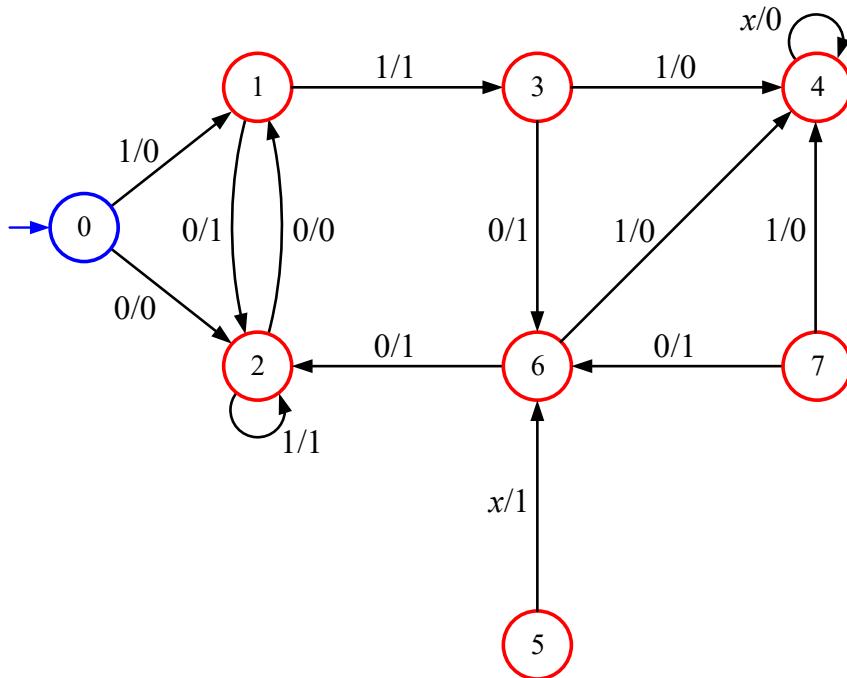


Figure A.20: A Mealy machine.

A second property that we want to verify is a *mutual-exclusion* property. A system satisfies the mutual-exclusion property if and only if it cannot be in more than one so-called *critical section* at the same time. Assume that our machine is a controller that controls a system with two critical sections such that the p -bit is true if and only if the system is in the first one of these critical sections, and the r -bit is true if and only if it is in the other one. The second property to be verified then is the following:

The controller ensures mutual exclusion.

1. Which of the two properties is true for our controller? Motivate your answer.
2. Give a DBA that can be used as a basis for the verification of these properties.

We come back to the actual verification of these properties in later exercises.
See answer.

A.5.2 LTL

The arbiter example shows that logic formulae for specifying properties, built from atomic propositions over an appropriately defined alphabet, form a natural combination with automata models of the system of interest, which define the behavior of the system as a language. Propositional logic itself is not sufficiently rich though to easily specify properties about words in such a language, or runs of these automata. To this end, we introduce

Linear-time Temporal Logic as an extension of propositional logic.

Syntax of LTL. LTL builds on propositional logic, adding two basic temporal operators, \circlearrowright ('next') and \circlearrowleft ('until'), and two derived temporal operators, \diamond ('eventually') and \square ('always').^a The basic LTL syntax is as follows:

$$\phi ::= \text{true} \mid \Pi \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \circlearrowright\phi \mid \phi_1 \circlearrowleft \phi_2$$

The derived syntax for propositional-logic formulae given in the previous subsection carries over to LTL. The additional derived temporal operators are defined as follows:

$$\begin{aligned}\diamond\phi &\stackrel{\text{def}}{=} \text{true} \circlearrowleft \phi \\ \square\phi &\stackrel{\text{def}}{=} \neg\diamond\neg\phi\end{aligned}$$

^aAn alternative, textual syntax often seen for LTL uses X for 'next' operator \circlearrowright , F ('finally') for 'eventually' operator \diamond and G ('globally') for 'always' operator \square .

Example A.27 (LTL). Consider again the arbiter of Example A.26. In that example and the accompanying Exercise A.31, four properties were mentioned. We can now precisely specify these properties:

Mutual exclusion $\square(\neg(grnt_1 \wedge grnt_2) \wedge \neg(grnt_1 \wedge grnt_3) \wedge \neg(grnt_2 \wedge grnt_3))$;

No waste $\square(grnt_1 \rightarrow req_1 \wedge grnt_2 \rightarrow req_2 \wedge grnt_3 \rightarrow req_3)$;

Service provision $\square((req_1 \vee req_2 \vee req_3) \rightarrow (grnt_1 \vee grnt_2 \vee grnt_3))$;

Fairness $\square\diamond(grnt_1 \vee \neg req_1) \wedge \square\diamond(grnt_2 \vee \neg req_2) \wedge \square\diamond(grnt_3 \vee \neg req_3)$.

Exercise A.34 (Properties of a Mealy machine). Consider the Mealy machine of Exercise A.32. Specify the mentioned property that an occurrence of o always results in the q bit being set as LTL formulae.

See answer.

Exercise A.35 (Properties of a controller). Consider the controller of Exercise A.33. Specify the two properties as LTL formulae.

See answer.

It is convenient to distinguish between basic syntax and derived syntax in the definition of logics, because this allows more compact definitions when developing the formalism, e.g., when providing semantics to the formulae and when converting between formalisms, like between formulae and automata. An LTL formula specifies an ω -language, that is defined for the basis LTL syntax as follows.

Suffix of infinite strings. For any string $\sigma \in \Sigma^\omega$, for some alphabet Σ , $\sigma[n..)$ is the suffix of σ starting at index n , i.e., for all $i \in \mathbb{N}$, $\sigma[n..)(i) = \sigma(n + i)$. Note that

$\sigma[n..)$ is itself an infinite string again.

Semantics of LTL. Let Σ be some alphabet of symbols. The following defines when a word $\sigma \in \Sigma^\omega$ satisfies LTL property ϕ , denoted $\sigma \models \phi$.

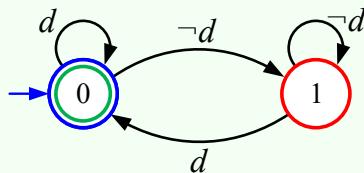
$$\begin{aligned}\sigma \models \text{true} & \\ \sigma \models p & \text{ iff } \sigma(0) \models p \quad (p \in \Pi) \\ \sigma \models \phi \wedge \psi & \text{ iff } \sigma \models \phi \text{ and } \sigma \models \psi \\ \sigma \models \neg\phi & \text{ iff } \text{not } \sigma \models \phi \\ \sigma \models \circlearrowleft \phi & \text{ iff } \sigma[1..) \models \phi \\ \sigma \models \phi \cup \psi & \text{ iff } (\exists n : n \in \mathbb{N} : \sigma[n..)) \models \psi \wedge (\forall i : 0 \leq i < n : \sigma[i..) \models \phi)\end{aligned}$$

Note that word σ satisfies an atomic proposition $p \in \Pi$ if and only if the first symbol in the word satisfies the atomic proposition (according to the interpretation of atomic propositions over symbols). Another important observation is that the semantics for the until operator requires that the second operand, ψ , is actually satisfied at some point. It cannot be postponed indefinitely.

An LTL formula ϕ now specifies ω -language $\mathcal{L}^\omega(\phi) = \{\sigma \in \Sigma^\omega \mid \sigma \models \phi\}$.

Example A.28 (LTL semantics). Consider the coffee machine of Figure A.10 again. Recall the alphabet $\Sigma = \{s, a, b, h, t, m, d, p\}$ of the coffee-machine actions. We ‘reuse’ this alphabet and overload notation to define atomic propositions q for any $q \in \Sigma$, formally denoting the singleton symbol set $\{q\}$.

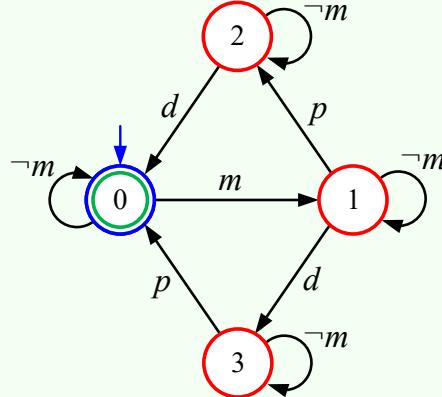
The property that the machine always again eventually serves coffee is then expressed by LTL formula $\phi = \square \diamond d$. A DBA accepting the language $\mathcal{L}^\omega(\phi)$ is the following:



This DBA expresses the fact that coffee is served (action d) infinitely often, allowing finite sequences of other actions in between. Note that we use the notational convenience to label transitions with propositional-logic formula $\neg q$ as a shorthand representing transitions for each of the symbols in $\Sigma \setminus \{q\}$.

The property of continuous proper service of the coffee machine, specified in Exercise A.16.3 as an ω -regular expression and stating that in each proper transaction coffee is properly made, served and paid for, can also be expressed in LTL. One could consider, for instance, the following formulation: $\square(m \rightarrow \circ(((\neg m) \cup d) \wedge ((\neg m) \cup p)))$. This phrasing explicitly requires that transactions do not overlap. No new coffee can be made until the last coffee made is served and paid for. So two m actions without p and d in between is considered a violation of the property. It was not asked to enforce this requirement in Exercise A.16.3. By replacing the $\neg m$ subformulae with `true` in the formulation, this requirement is no longer enforced.

An NBA over alphabet Σ accepting the language corresponding to the given LTL property is the following:



To facilitate reasoning with LTL formulae, we consider the following natural notion of equivalence of formulae.

LTL equivalence. Two LTL formulae ϕ and ψ are equivalent, denoted $\phi \equiv \psi$, if and only if the formulae specify the same language, $\mathcal{L}^\omega(\phi) = \mathcal{L}^\omega(\psi)$.

LTL equivalences. Let ϕ , ψ , and ξ be LTL formulae. The following are useful equivalences:

| | | | |
|--------------------|--|---------------------|---|
| Duality | $\neg \Box \phi \equiv \Diamond \neg \phi$ | Distribution | $\Box(\phi \wedge \psi) \equiv \Box \phi \wedge \Box \psi$ |
| | $\neg \Diamond \phi \equiv \Box \neg \phi$ | | $\Diamond(\phi \vee \psi) \equiv \Diamond \phi \vee \Diamond \psi$ |
| | $\neg \Diamond \phi \equiv \Box \neg \phi$ | | $\Diamond(\phi \wedge \psi) \equiv \Diamond \phi \wedge \Diamond \psi$ |
| Idempotence | $\Box \Box \phi \equiv \Box \phi$ | | $\Diamond(\phi \vee \psi) \equiv \Diamond \phi \vee \Diamond \psi$ |
| | $\Diamond \Diamond \phi \equiv \Diamond \phi$ | | $\Diamond(\phi \mathbf{U} \psi) \equiv (\Diamond \phi) \mathbf{U} (\Diamond \psi)$ |
| | $\phi \mathbf{U} (\phi \mathbf{U} \psi) \equiv \phi \mathbf{U} \psi$ | | $\xi \mathbf{U} (\phi \vee \psi) \equiv (\xi \mathbf{U} \phi) \vee (\xi \mathbf{U} \psi)$ |
| | $(\phi \mathbf{U} \phi) \mathbf{U} \psi \equiv \phi \mathbf{U} \psi$ | | $(\phi \wedge \psi) \mathbf{U} \xi \equiv (\phi \mathbf{U} \xi) \wedge (\psi \mathbf{U} \xi)$ |
| Absorption | $\Box \Diamond \Box \phi \equiv \Diamond \Box \phi$ | Expansion | $\Box \phi \equiv \phi \wedge \Box \phi$ |
| | $\Diamond \Box \Diamond \phi \equiv \Box \Diamond \phi$ | | $\Diamond \phi \equiv \phi \vee \Diamond \phi$ |
| | $\phi \vee \Diamond \phi \equiv \phi$ | | $\phi \mathbf{U} \psi \equiv \psi \vee (\phi \wedge \Box(\phi \mathbf{U} \psi))$ |
| | $\phi \wedge \Box \phi \equiv \Box \phi$ | | |

The expansion equivalences are of particular interest. These show how a formula can be written in the form of non-temporal parts that need to hold for the first symbol of a word and temporal parts using the next operator that need to hold for the following symbols in the word. In line with Example A.28, any LTL formula can be transformed into a generalized NBA as defined in Definition A.17. The conversion from LTL properties to GNBA is based on expansion of formulae. The precise conversion is tedious though. Also the resulting GNBA is not always intuitive, as we have also seen that the translations between regular

expressions and automata do not always yield intuitive and the most compact results. Since the most important goal of the material in these notes is to gain insight in languages, logics, and automata, we omit the precise conversion from LTL to GNBA. The interested reader is referred to [3]. In the exercises below, we rely on insight to convert from LTL properties to GNBA that accept the same language. Expansion of LTL formulae may then be a helpful concept.

Example A.29 (Expansion, conversion from LTL to (G)NBA). Reconsider Example A.25 about checking ω -regular properties. In that example, the bad behaviors were characterized by all executions with an infinite tail without r_1 action. In LTL, the bad behaviors can then be defined as follows:

$$\Diamond \Box \neg r_1$$

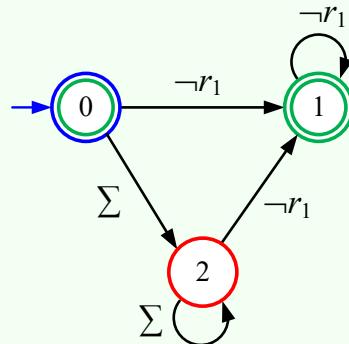
We can apply expansion twice and absorption to obtain the following:

$$\begin{aligned} \Diamond \Box \neg r_1 &\equiv \Box \neg r_1 \vee \Diamond \Box \neg r_1 \\ &\equiv \neg r_1 \vee \Box \neg r_1 \vee \Diamond \Box \neg r_1 \\ &\equiv \neg r_1 \vee \Diamond (\Box \neg r_1 \vee \Diamond \Box \neg r_1) \\ &\equiv \neg r_1 \vee \Diamond \Box \neg r_1 \\ &\equiv (\neg r_1 \wedge \text{Otrue}) \vee (\text{true} \wedge \Diamond \Box \neg r_1) \end{aligned}$$

One more application of expansion on the subformula $\Box \neg r_1$ gives the following:

$$\Box \neg r_1 \equiv \neg r_1 \wedge \Diamond \Box \neg r_1$$

These results suggest an automaton with three states, one representing the initial formula $\Diamond \Box \neg r_1$ (state 0 in the NBA below), one representing $\Box \neg r_1$ (state 1) and one representing **true** (state 2). The transitions correspond to atomic propositions $\Sigma = \{r_1, p_1, \neg p_1, r_2, p_2, \neg p_2\}$ and $\neg r_1$ as shorthand for $\Sigma \setminus \{r_1\}$.



The state corresponding to the original formula is initial and final. Whether or not the states resulting from a transition are final, depends on the atomic proposition corresponding to the transition. In this example, a $\neg r_1$ transition implies that the resulting state is final. The automaton can be simplified to a two-state automaton, by merging states 0 and 2, making the merged state non-final. This results in fact in the NBA already given in Example A.25.

Exercise A.36 (Properties of the coffee machine - expansion). Consider the two properties of our coffee machine specified in Example A.28. Expand the given formulae to a non-temporal part that needs to hold for the first symbol of any word in the language of the property and a temporal part that needs to hold for all following symbols.

See answer.

Exercise A.37 (Properties of a Mealy machine - automata). Consider the Mealy machine of Exercise A.32 and its property specified in Exercise A.34. Expand the LTL formula of Exercise A.34 to a non-temporal part that needs to hold for the first symbol of any word in the language of the property and a temporal part that needs to hold for all following symbols. Give a (G)NBA accepting the language specified by the property.

See answer.

Exercise A.38 (Properties of a controller - automata). Consider the controller of Exercise A.33 and its properties specified in Exercise A.35. Expand the LTL formulae of Exercise A.35. Give (G)NBA accepting the languages specified by the two properties.

See answer.

Exercise A.39 (Properties of the arbiter - automata). Consider the arbiter properties specified in Example A.27. Expand these formulae. Give (G)NBA accepting the languages specified by these properties. For the fairness property, consider one and-clause (i.e., fairness of one processor) at a time.

See answer.

A.5.3 Checking LTL properties

Checking LTL properties for systems specified through ω -languages goes along the same lines as checking ω -regular properties. See Algorithm A.10 below. As in Algorithm A.9, the negation of property of interest ϕ serves as a starting point for the check, which then essentially consists of an emptiness check of a product language.

Algorithm A.10 (Checking LTL properties). Let ϕ be an LTL property specifying the property of interest. Let S be an ω -regular language describing all system behaviors. It can be verified as follows whether $S \models \phi$:

1. Create a GNBA $B_{\neg\phi}$ specifying the bad behaviors $\mathcal{L}^\omega(\neg\phi)$;
2. create a GNBA B_S for S ;
3. create the product GNBA $B_S \otimes B_{\neg\phi}$;

4. check for emptiness of $\mathcal{L}^\omega(B_S \otimes B_{\neg\phi})$.

If the language $\mathcal{L}^\omega(B_S \otimes B_{\neg\phi}) = \emptyset$, then $S \models \phi$, i.e., system S validates property ϕ .

Exercise A.40 (Checking properties of a Mealy machine). Consider the Mealy machine of Exercise A.32 and its property specified in Exercises A.34 and A.37. Verify whether or not the Mealy machine satisfies the property following the steps of Algorithm A.10. Start by specifying the bad behaviors as an LTL formula. Then create a (G)NBA capturing the bad behaviors, give a product GNBA, and check for emptiness. See answer.

Exercise A.41 (Checking properties of a coffee machine). Consider the coffee machine and its properties discussed in Example A.28. Verify whether or not the coffee machine satisfies the two properties following the steps of Algorithm A.10. Start by specifying the bad behaviors of each property as an LTL formula. Then create a (G)NBA capturing the bad behaviors, give a product GNBA, and check for emptiness for each of the properties.

See answer.

Exercise A.42 (Checking properties of a controller). Consider the controller of Exercise A.33 and its properties specified in Exercises A.35 and A.38. Verify whether or not the controller satisfies the properties by creating a product GNBA and checking for emptiness in line with Algorithm A.10.

See answer.

Exercise A.43 (Checking properties of the arbiter). Consider the arbiter of Example A.26 and its properties of Example A.27. Show that the arbiter does not satisfy the fairness property by creating a product GNBA and checking for emptiness in line with Algorithm A.10. Adapt the Mealy model of the arbiter so that it becomes a fair arbiter and show that the adapted model satisfies the four properties of Example A.27. Hint: consider fairness of the three processors individually.

See answer.

As mentioned, any LTL formula can be converted into a GNBA accepting the same ω -language. The expressiveness of LTL is therefore limited to ω -regular properties. LTL cannot express all ω -regular languages though, see e.g., [3, 11]. Thus, LTL is less expressive than ω -regular expressions and NBA. So why are we interested in LTL, and other temporal logics? The key reasons are that temporal logics often allow intuitive specifications of properties and more efficient, specialized implementations of property checking than property checking on general GNBA.

A.6 Answers to Exercises

Exercise A.1 (Words and languages).

1. L_5 is the language of all words in which every 1 is immediately followed by a 0. Thus, “00” and “0100” are elements of L_5 , the former because the condition that every 1 is followed by a 0 holds if a word does not contain a 1. The words “0110” and “01010001” are not elements of L_5 , the latter because the last symbol is a 1 that is not followed by a 0.
2. L_6 is the language of all words in which every 1 is immediately followed by another 1. Only words not containing any 1s are elements of L_6 . Any word with a 1 has at least one 1, namely the last one, that is not followed by a 1. Thus, from the mentioned words, only “00” is an element of L_6 .
3. L_7 contains all words with an even number of 1s. So, ε , “00”, and “0110” are elements of L_7 and “0100” and “01010001” are not.
4. We can define L_8 by counting the number of 1s and the number of 0s in a word, and equating them: $L_8 = \{\sigma \in \Sigma^* \mid (+i : i \in \mathbb{N} \wedge \sigma(i) = 0 : 1) = (+i : i \in \mathbb{N} \wedge \sigma(i) = 1 : 1)\}$.

Exercise A.2 (Operations on languages). Language concatenation is not commutative. Consider, for instance, alphabet $\{a, b\}$ with languages $\{a\}$ and $\{b\}$. Then, $\{a\} \cdot \{b\} = \{ab\} \neq \{ba\} = \{b\} \cdot \{a\}$.

Exercise A.3 (Regular expressions).

1. Language $\mathcal{L}(\beta\alpha\alpha^*(E\alpha\alpha^*))$ differs from L_2 , because it does not contain words without an E occurrence. For instance, -3 is not an element of the language, whereas it is an element of L_2 .
2. Language $\mathcal{L}(\beta\alpha\alpha^*(E\alpha\alpha)^*)$ differs from L_2 because the number of digits after an E occurrence is always two and because it is possible to have multiple E s in a word. E.g., +3E10E10 is a possibility and -3E2 is not.
3. Also language $\mathcal{L}(\beta\alpha\alpha^*(E\alpha\alpha^*)^*)$ differs from L_2 , again because multiple occurrences of E are possible. For instance, -3E2E2 is allowed, whereas it is not part of L_2 .
4. The definition of language union implies that $\mathcal{L}(\beta\alpha\alpha^*(\emptyset + E\alpha\alpha^*)) = \mathcal{L}(\beta\alpha\alpha^*(E\alpha\alpha^*))$. That is, the language defined by the expression at hand is equal to the language defined by the first expression in this exercise. Hence, also in this case, the language differs from L_2 .
5. The first and fourth expression define the same language, as already argued. The second expression defines a language that is different from all others because it is the only expression that only allows E occurrences in combination with two subsequent digits. The third expression defines a language that is different from all others because it is the only language containing the mentioned example word -3E2E2.

6. Expression $a(a^* + b^*)$ does not allow to switch between a -s and b -s after the initial a , whereas $a(a + b)^*$ does. So, aba is an element of the language of $a(a + b)^*$ but not of the language of $a(a^* + b^*)$.
7. The two expressions define the same language, because the nesting of repetitions of a in the first expression does not add or remove any options compared to the repetition in the second expression.

Exercise A.4 (Creating regular expressions). Assume that regular expression $\beta = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ and expression $\alpha = 0 + \beta$. In all of the following cases, multiple correct answers are possible.

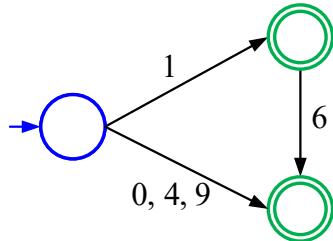
1. $(\alpha^* 1)^* \beta^*$.
2. $\gamma (+\gamma)^* + (\gamma (+\gamma)^*) = (\gamma (+\gamma)^*)$ with $\gamma = 0 + \beta \alpha^*$.
3. $(0 + 10)^*$.
4. 0^* .
5. $(0 + 10^* 1)^*$.
6. It is not possible to give a regular expression for this language. There is no means to count the number of 0s and 1s so that they can be kept equal, without explicitly enumerating options. It is not possible to enumerate infinitely many options in a finite expression. Languages that cannot be specified with a regular expression are investigated later in these course notes.

Exercise A.5 (Languages of finite automata).

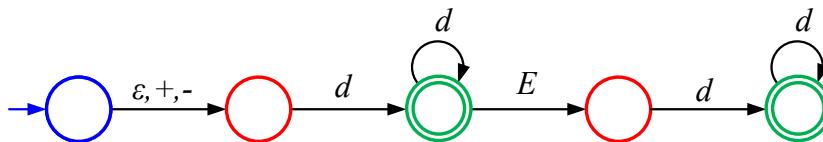
1. Let A be the automaton of Example A.4.
 - $\mathcal{L}(A)$ contains all words in which every a (if any) is always at some point followed by a b ;
 - $\mathcal{L}(A) = \{\sigma \in \Sigma^* \mid (\forall i : i \in \mathbb{N} \wedge \sigma(i) = a : (\exists j : j \in \mathbb{N} \wedge j > i : \sigma(j) = b))\}$;
 - $\mathcal{L}(A) = \mathcal{L}((b + c + a \cdot (a + c)^* \cdot b)^*)$.
2. Both automata do not define the same language as the automaton of Example A.5. Both accept, for instance, a word $.+3$, which is not accepted by the automaton of Example A.5. The second automaton moreover accepts words with multiple E -s, which are not accepted by the other two automata.

Exercise A.6 (Creating automata models).

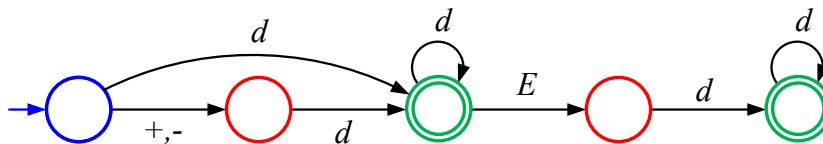
1. The following automaton accepts language L_1 of Example A.1. It is deterministic.



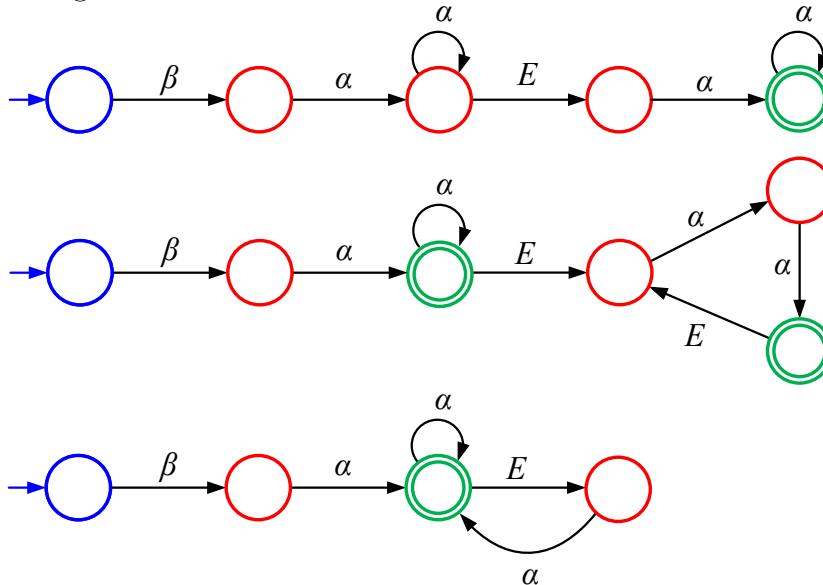
The following automaton accepts language L_2 of Example A.1. It is not deterministic, because of the ε move enabled in the initial state.



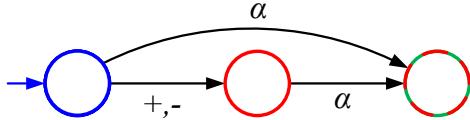
A DFA accepting the same language is the following:



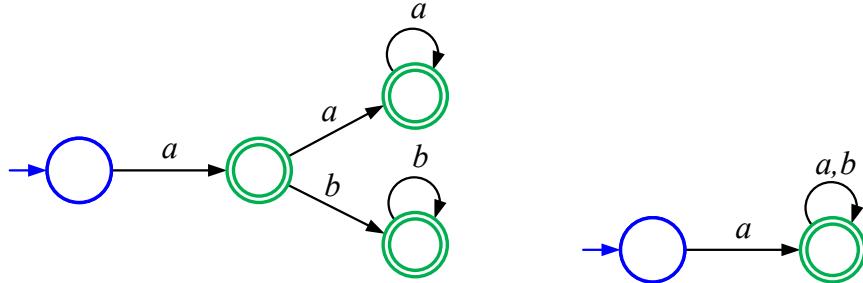
2. The languages of the first three expressions of Exercise A.3 are recognized by the following three automata:



The language of the fourth expression is again recognized by the first of these three automata. None of these automata are deterministic, because they all allow an ε move in the initial state. The automata can be made deterministic following the same pattern as in the previous item:



Automata for the expressions $a(a^* + b^*)$ and $a(a + b)^*$ ($= a(a^* + b)^*$) are the following:

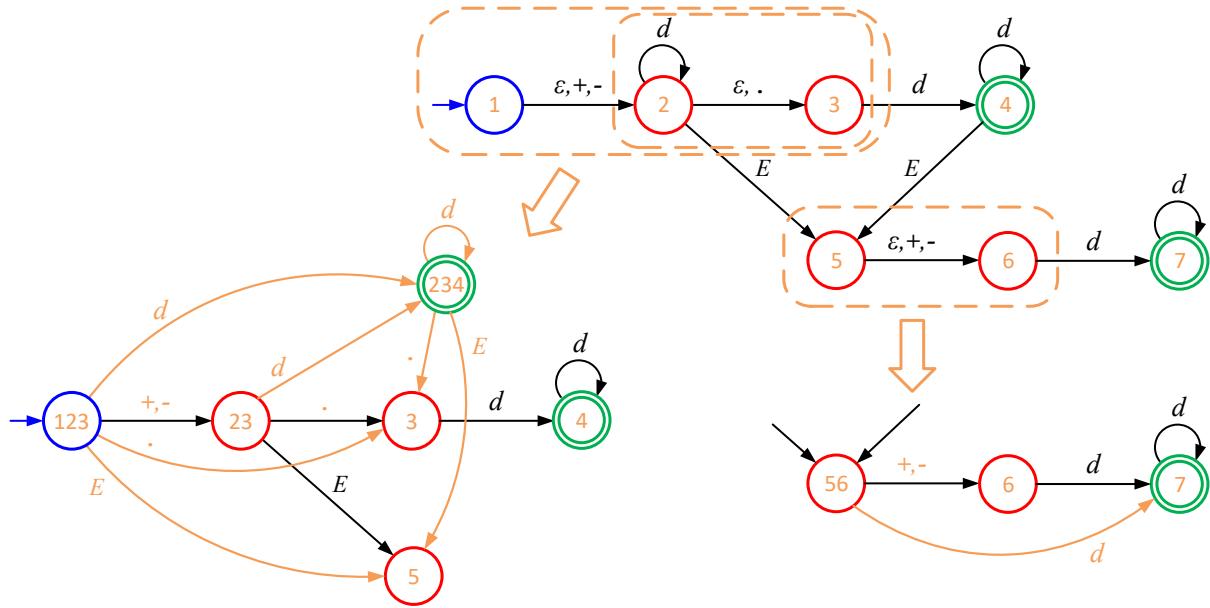


Both automata are deterministic.

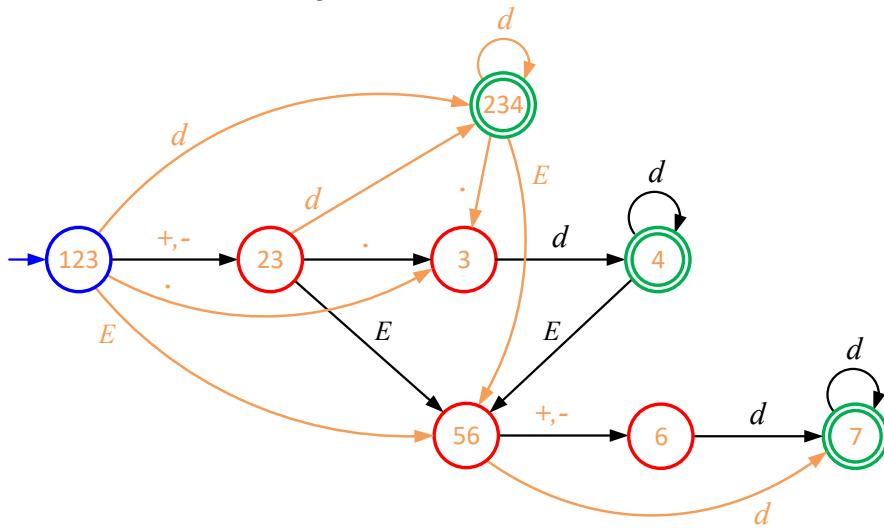
3. Use the CMWB to practice with automata:

- Create an automaton YA for the language at hand.
- Enter your automaton YA in the CMWB.
- Take the corresponding regular expression of Exercise A.4 and enter it in the CMWB.
- Convert the expression to a finite automaton EA .
- Check language inclusion $\mathcal{L}(YA) \subseteq \mathcal{L}(EA)$ between your automaton and the expression automaton.
- Check language inclusion $\mathcal{L}(EA) \subseteq \mathcal{L}(YA)$ between the expression automaton and your automaton.
- If both inclusions hold, then $\mathcal{L}(EA) = \mathcal{L}(YA)$ and hence your answer is correct.
- If any of the inclusions does not hold, then learn from the counter examples and retry.

4. The automaton of Figure A.2 has several ε moves that make the automaton nondeterministic. To create a DFA accepting the same language, we need to apply the pattern shown in the first two items of this exercise several times. It is important to observe though that, from the initial state, two subsequent ε moves are possible. Intuitively, this means that the automaton could ‘start’ in any of the first three states. Thus, all the possible behaviors without ε that this allows, need to be reconstructed in the DFA. The construction is as follows:



This leads to the following DFA:

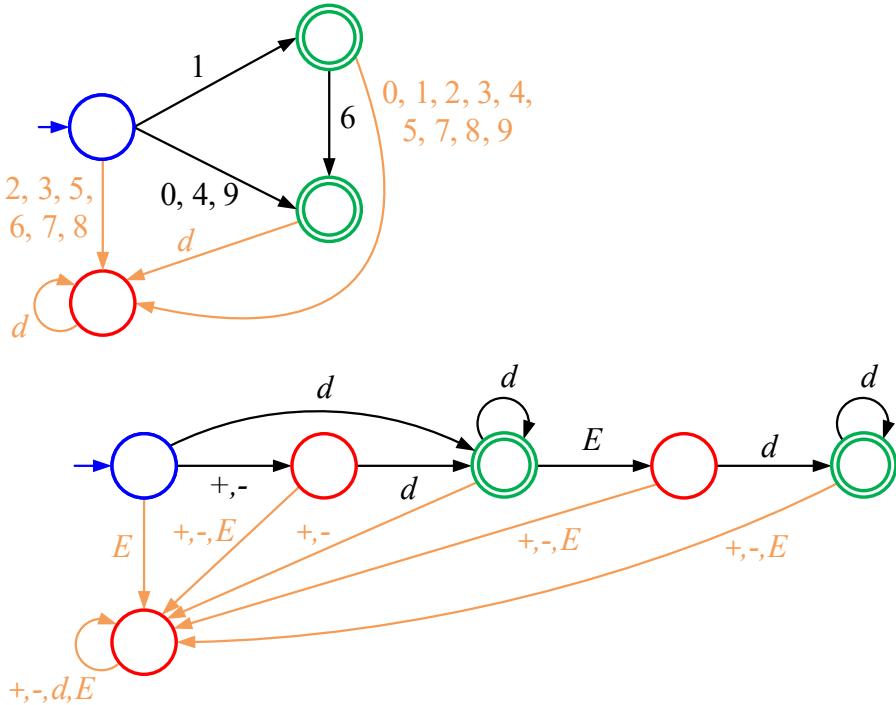


Section A.2.4 presents a systematic means of transforming an NFA- ε to an equivalent DFA.

5. It is not possible to define a finite automaton for the language $L8$ of Exercise A.1. When recognizing a word, such an automaton would need to keep track of the difference between the numbers of encountered 0s and encountered 1s, because only if the remainder of the word being recognized compensates for that difference, the word can be accepted. But there are infinitely many possible differences (any natural number is possible). These differences can only be differentiated by the automaton through different states. But this is therefore not possible with finitely many states. Since an NFA can only have finitely many states, such an NFA cannot exist. The expressiveness of automata is further elaborated in Section A.2.3.

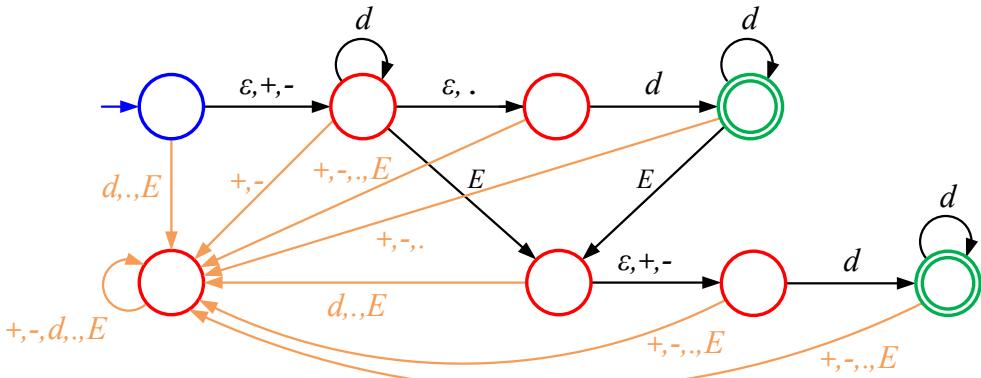
Exercise A.7 (Completing automata models).

1. The two DFA given for languages L_1 and L_2 in the answer to Exercise A.6.1 are both incomplete. The following are two complete DFA that accept the same languages (where d is an abbreviation for any digit in $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$).



For all the other DFA in your answers, you may check in the CMWB whether your completed DFA and the original DFA that you started with accept the same language (by checking language inclusion twice, as explained in the answer to Exercise A.6.3 above).

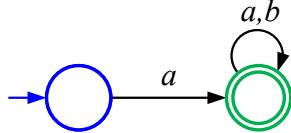
2. The completed automaton is as follows:



6.7 and +4.5 are two words in the language of the automaton of Figure A.2 with only one accepting run in that automaton, that both have also a run to the newly added state in the above completed automaton. This illustrates that the nondeterminism in the automaton increased.

Exercise A.8 (Complement automata).

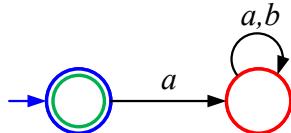
1. Consider alphabet $\Sigma = \{a, b\}$ and the following automaton A , with $\mathcal{L}(A) = \{\sigma \in \Sigma^+ \mid \sigma(0) = a\}$:



That is, A accepts all words starting with an a .

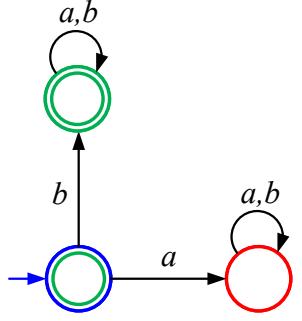
The complement language $\overline{\mathcal{L}(A)} = \{\sigma \in \Sigma^* \mid \sigma(0) \neq a\}$ is the language of all words not starting with an a , including the empty word ε and words like b , bb , and $baba$.

If we apply the complement operation to A without first completing it, we get the following automaton \bar{A} :



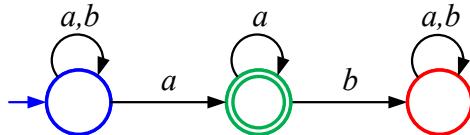
It easily follows that $\mathcal{L}(\bar{A}) = \{\varepsilon\} \neq \overline{\mathcal{L}(A)}$.

If we first complete A to get cA and then complement cA to \overline{cA} ,



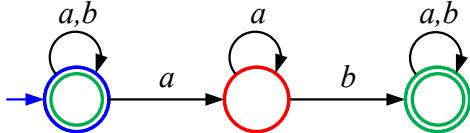
then we obtain the expected languages $\mathcal{L}(\overline{cA}) = \overline{\mathcal{L}(A)}$.

2. Consider alphabet $\Sigma = \{a, b\}$ and the following complete NFA cA :



That is, $\mathcal{L}(cA)$ contains all words that end with an a . The expected complement language $\overline{\mathcal{L}(cA)}$ is the language of all words not ending with an a , including the empty word ε and words like b , $abab$.

If we complement cA to \overline{cA}



though, then we get an incorrect result. $\mathcal{L}(\overline{cA})$ contains many words ending with an a . In fact, $\mathcal{L}(\overline{cA}) = \Sigma^* \neq \overline{\mathcal{L}(cA)}$.

Exercise A.9 (Pumping lemma).

1. (a) For language L_2 , 5 is a valid pumping length. Any word in L_2 with length at least 5 has a two-digit subword. One of these digits can be dropped or pumped. Examples are $+1E23$ or $456E7$. Subwords that can be pumped for these two examples are 2 or 3, and 4, 5, 6, 45, or 56, respectively. $+1E3$, $+1E2223$, $6E7$, and $45456E7$, are, for example, all elements of L_2 .
A smaller pumping length does not exist, since, e.g., $+1E2$ is a word of length 4 that does not have a part that can be repeated arbitrarily often, including zero (!) times.
- (b) Assuming regular expressions $\alpha = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ and $\beta = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$, L_3 is specified by e.g. regular expression $(\alpha^*1)^*\beta^*$. This expression suggests that any part of any word in the language can be repeated arbitrarily often. So 1 is a valid pumping length. For any non-empty word in L_3 , the first digit can be repeated arbitrarily often, including zero (!) times, resulting again in a word in L_3 . Examples are 01, leading to 1 when the 0 is dropped and with pumped words 0001 and 0000001, and 43, leading to words like 3, 443, and 44444443.
- (c) The longest word in L_4 that has no repeatable subword is of length 3, for instance word $0 = 0$. 4 is a valid pumping length. Any word of length at least 4 has either a sequence of two digits or a subword $a+b$ with a and b digits in its first four symbols. One of these two digits or one of the summands of the addition (including the + symbol) can be dropped/pumped. An example is 1234, where 4 can be dropped or pumped, giving accepted words such as 123, 123444 and 12344444. In $1 + 0 = 0$, the $+0$ subword can be dropped or pumped giving e.g. $1 = 0$ and $1 + 0 + 0 + 0 = 0$, which are all part of L_4 .
- (d) L_5 is specified by e.g. regular expression $(0 + 10)^*$. Any word in L_5 is built from zero or more 0 symbols and 10 symbol pairs. So 2 is a valid pumping length. Any initial 0 can be dropped/pumped. For 00, e.g., 0, 0000, and 0000000 are examples of versions of 00 in which the initial 0 is dropped/pumped. Similarly, for 010, we get 10, 00010, and 00000010. Also any initial 10 can be dropped/pumped: 100 leads to e.g., 0, 1010100, and 1010101010100.
- (e) L_6 is specified by regular expression 0^* . So 1 is a valid pumping length. For 0, for example, 0000 and 0000000 are examples of pumped versions of 0; dropping 0 leads to ϵ , which is also an element of L_6 .
- (f) L_7 is specified by $(0 + 10^*1)^*$. Any accepted word of length at least 2 has a repeatable/droppable subword in its first two symbols. Any initial 0 can be

dropped/pumped, with 011, 11, 000011, and 000000011 examples of a word and versions of that word resulting from dropping/pumping the initial 0. Also any initial 11 can be dropped/pumped, e.g., 110, 0, 1111110, and 11111111110 are all elements of L_7 . And the 0 in e.g. 101 can be dropped or pumped, leading to e.g. 11 and 100000001 that are elements of L_7 . Since these cases cover all possible prefixes of length 2 of accepted words, 2 is a valid pumping length.

2. Proving that language L_8 is not regular follows the reasoning of Example A.8 with 0 and 1 replacing a and b .
3. The language of properly nested pairs is not regular. Again, the proof follows the reasoning of Example A.8 with { and } instead of a and b .

Exercise A.10 (Pigeonhole principle). Proving non-regularity of any of the two languages in this exercise using the Pigeonhole principle follows exactly Example A.9, based on the observation that any word $a^n b^n$ with $a = 0$, $b = 1$ resp. $a = \{$, $b = \}$ for any $n \in \mathbb{N}$ is an element of the language.

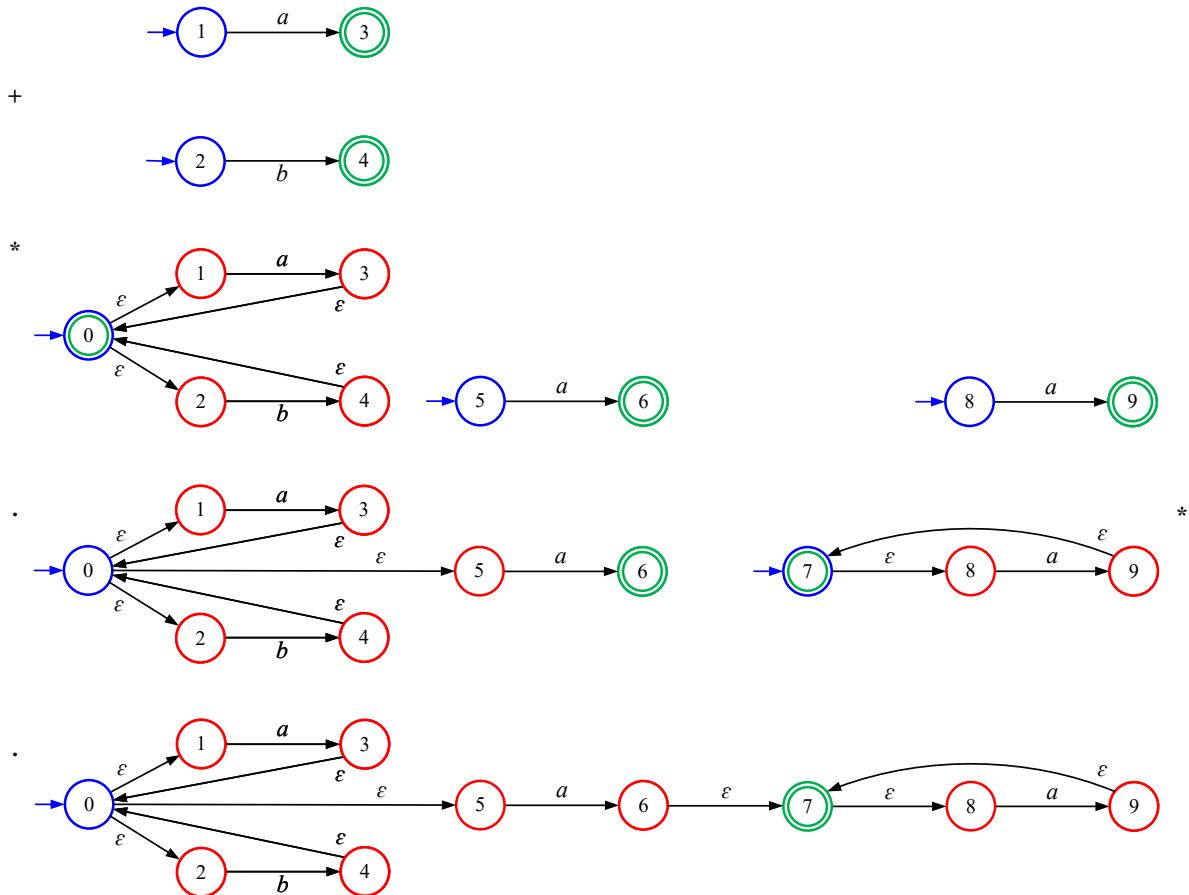
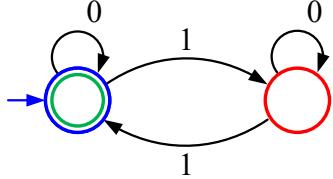


Figure A.21: Converting $(a + b)^*aa^*$ to an automaton.

Exercise A.11 (Conversion from a regular expression to an NFA- ε). Figure A.21 shows the conversion, starting from four primitive automata, and then applying the conversions for $+$, $*$ (twice), and \cdot (twice). The operations are shown in the left and right margins.

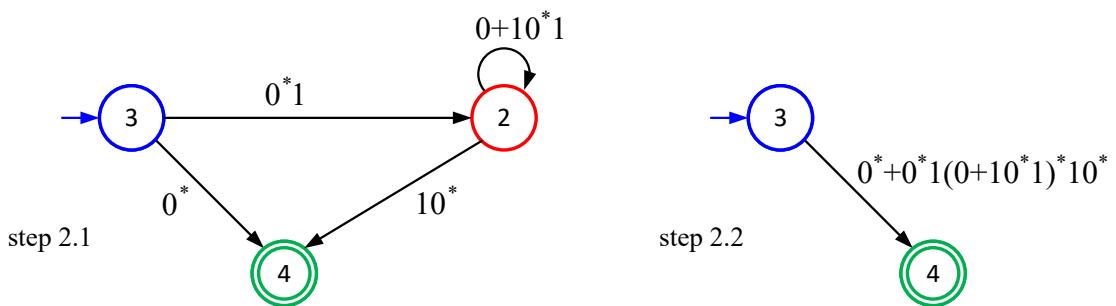
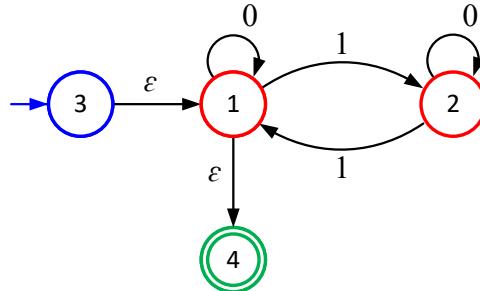
Exercise A.12 (Conversion from an NFA- ε to a regular expression, state elimination). We provide examples of state elimination by including the answer for language $L7$ in the first item of the exercise, because it is a relatively straightforward example of state elimination, and for the automaton of Figure A.2 in the second item, because this is a more involved conversion. For the other languages in the first item, you may use the CMWB to check your answer. Enter your expression in the CMWB, convert it to an automaton, and check equivalence of this automaton and the original automaton you started with (by checking language inclusion twice).

1. A DFA recognizing $L7$ is the following.



State elimination then goes as follows:

automaton in the right form,
after step 1



with $0^* + 0^*1(0 + 10^*1)^*10^*$ as the final result.

2. Figure A.22 shows the state elimination, with the following expression as a result:

$$\beta\alpha^*(\varepsilon + .)\alpha\alpha^* + (\beta\alpha^*E + \beta\alpha^*(\varepsilon + .)\alpha\alpha^*E)\beta\alpha\alpha^*.$$

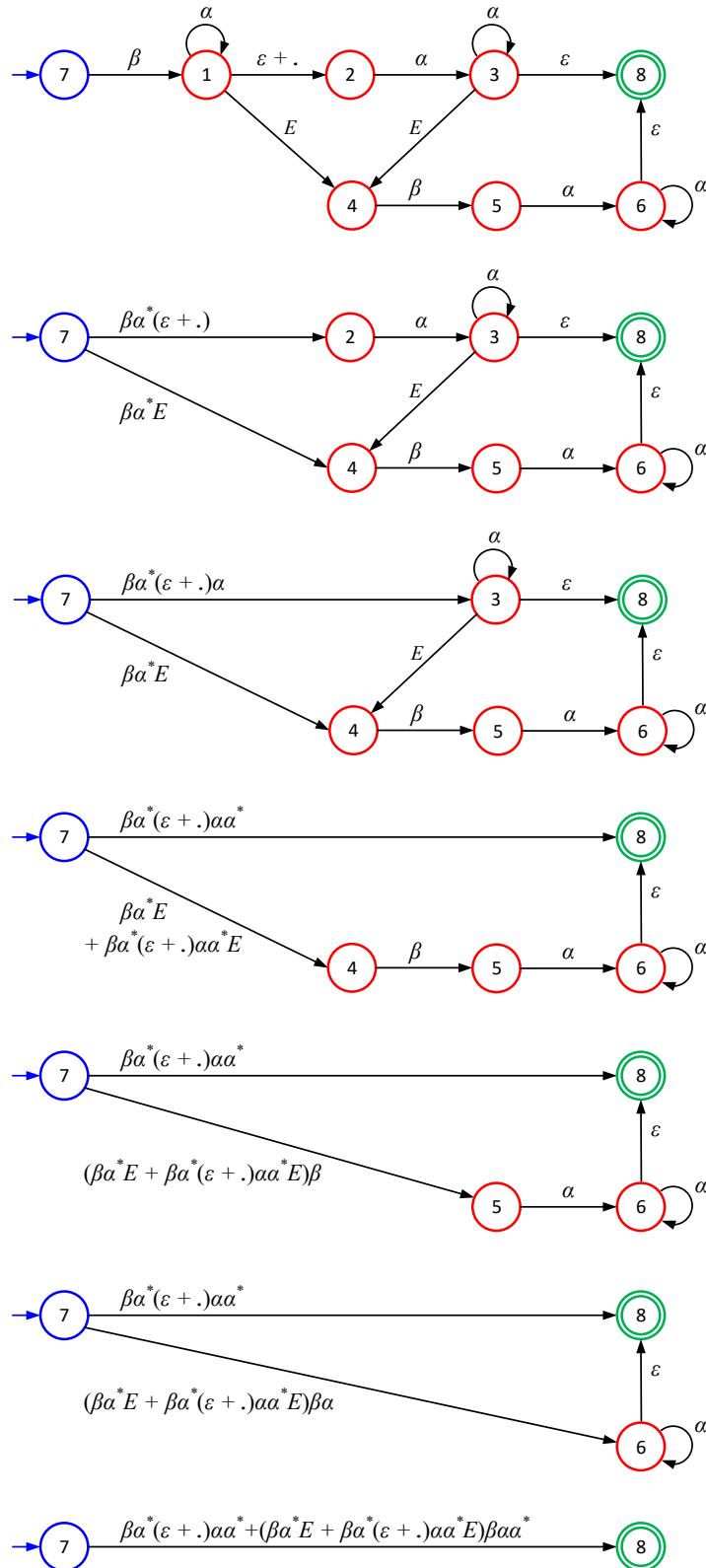
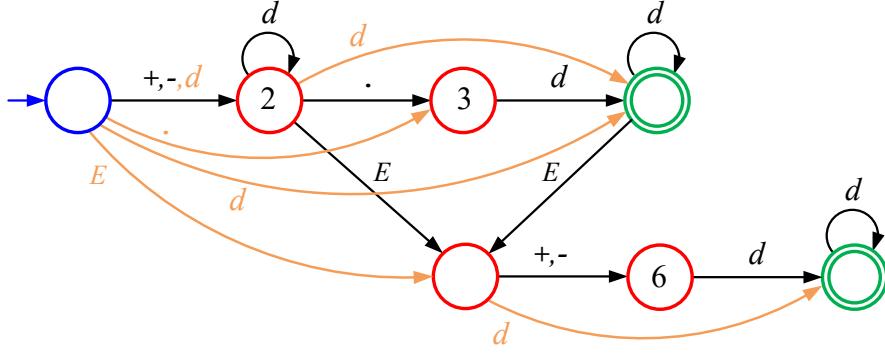


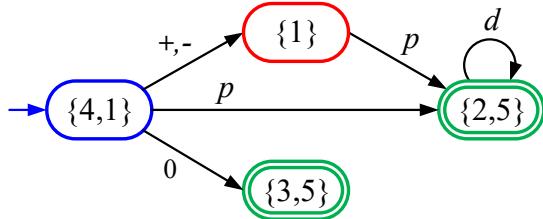
Figure A.22: Converting the automaton of Figure A.2 to a regular expression via state elimination.

Exercise A.13 (Conversion from an NFA- ε to an NFA). The conversion gives the automaton shown below. The original automaton has three ε moves, to the states numbered 2, 3, and 6 in the automaton below. We need to add six new transitions, shown in orange, for the three transitions leaving state 2 and the two transitions leaving states 3 and 6.



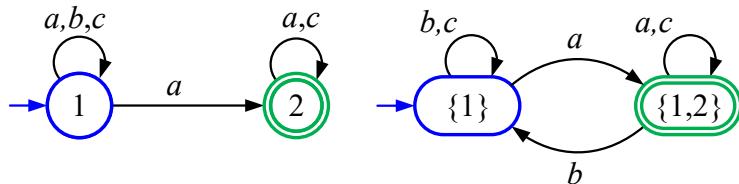
Exercise A.14 (Conversion from an NFA- ε to a DFA). The conversion results for the first four items of this exercise are given below. You may use the CMWB to check language equivalence between the starting automata and your answers to verify the correctness of your answers.

1. The conversion leads to the following result:

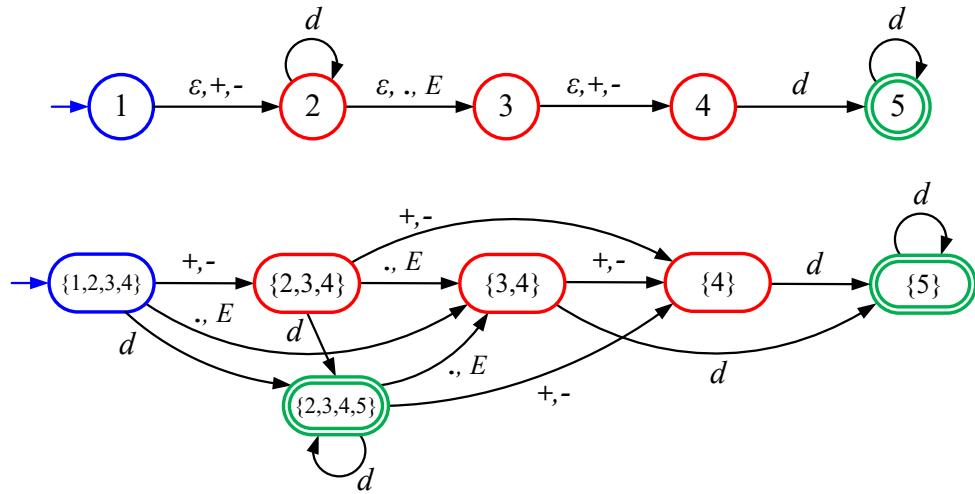


The DFA is identical to the DFA of Figure A.8, except for the state names. States {2} and {3} in Figure A.8 are extended with the extra state 5 of the NFA, which can be reached from 2 and 3 through an ε move.

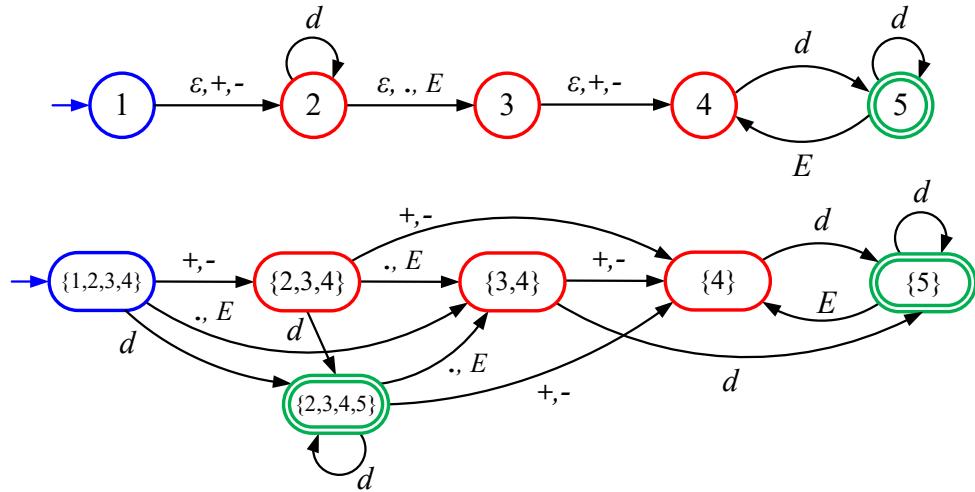
2. The following shows the automaton of Figure A.3(a) with numbered states and the equivalent DFA resulting from the subset construction.



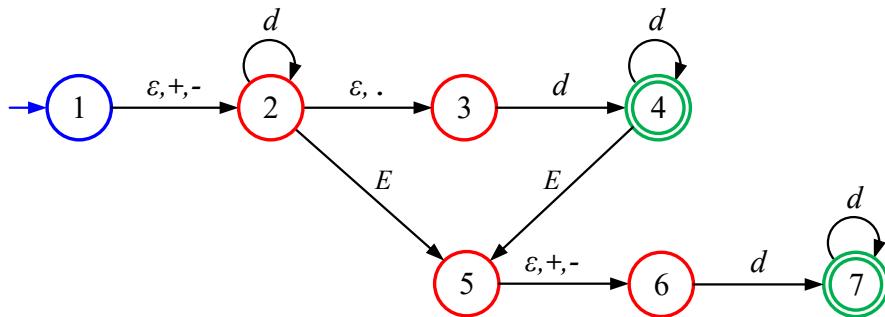
3. The automata of Exercise A.5.2 and their equivalent DFA are as follows:



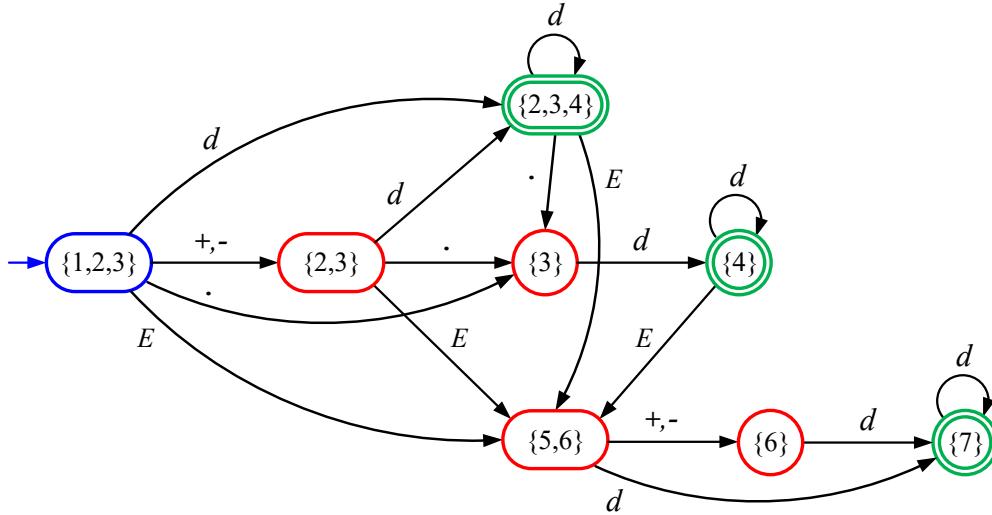
and



4. The automaton of Figure A.2 and the DFA are as follows:



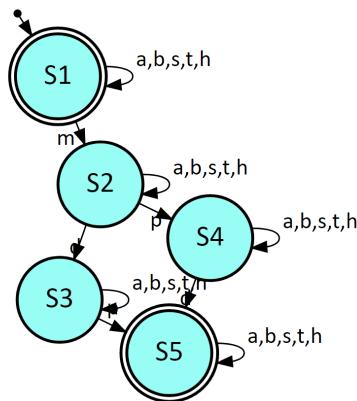
and



5. Check your answers with the CMWB and/or generate DFA directly with the CMWB.

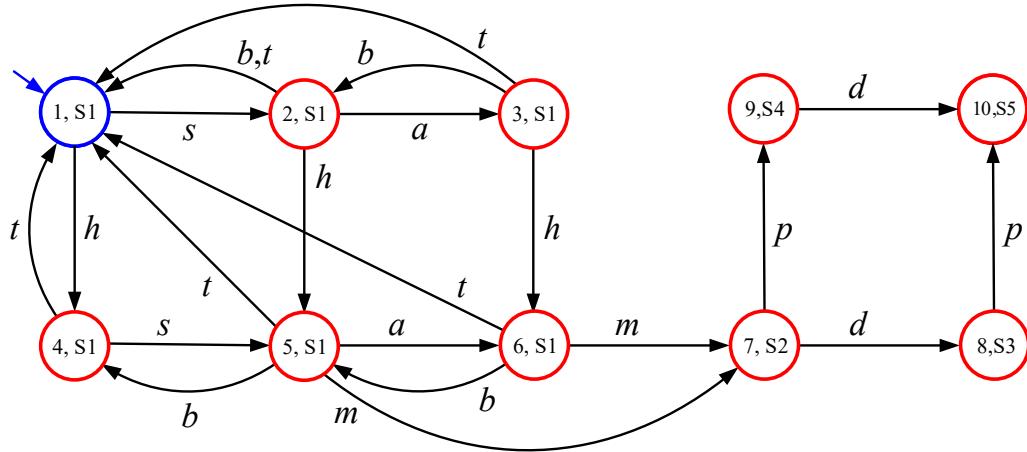
Exercise A.15 (Checking regular properties).

1. The correct property, phrased as a regular expression, is $\alpha^*(\varepsilon + m\alpha^*(d\alpha^*p + p\alpha^*d)\alpha^*)$ with $\alpha = s+a+b+h+t$. You can verify the steps and your answers using the CMWB, through the following steps, if you start from this regular expression:
 - (a) Enter the coffee-machine model in the CMWB.
 - (b) Enter the property in the workbench and convert it to an NFA.
 - (c) Convert the result to a DFA, minimize, and relabel. The result may then look as follows:



You may also start with a DFA for your property directly, which should then be as above.

- (d) Complete the DFA. This leads to a DFA with one extra state, following the examples given earlier in the course notes.
- (e) Complement the complete DFA, swapping initial and final states.
- (f) Take the product with the coffee machine DFA. The result follows the DFA of the coffee machine, but without any final states:



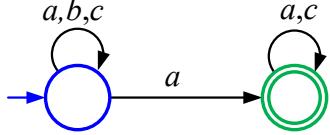
- (g) Check for emptiness. Because the product does not have any final states, it is clear that the language is indeed empty, indicating that the property is valid.
2. In the first item of this exercise, you have shown that every behavior of the machine that delivers a drink also processes a payment. So it is not possible to cheat the coffee machine and get a free drink.

Exercise A.16 (ω -Regular languages, ω -regular expressions).

1. (a) $((\alpha^* \cdot 1)^* \cdot \beta^*)^\omega$ also correctly specifies $L3^\omega$.
 - (b) $(\alpha^* \cdot 1 \cdot \beta)^\omega$ contains only words that have infinitely many 1-s. So a word like 9^ω is an element of $L3^\omega$ but not of the language defined by $(\alpha^* 1 \beta)^\omega$.
 - (c) The language specified by $(\alpha^* \cdot 1)^* \cdot \beta^\omega$ allows only finitely many 0-s in any accepted word. So a word like $(01)^\omega$ is not accepted, whereas this word is an element of $L3^\omega$.
 - (d) The language specified by $(\alpha^* \cdot 1 \cdot \beta^*)^\omega + \beta^\omega$ does not contain words like 019^ω that contain finitely many 0-s and 1-s that satisfy the requirements of $L3^\omega$. But such a word is an element of $L3^\omega$. The first option of the given expression $((\alpha^* 1 \beta^*)^\omega)$ requires infinitely many 1-s; the second option (β^ω) does not allow any 0-s.
 - (e) The language specified by $(\alpha^* \cdot 1)^* \cdot \beta^\omega + \beta^\omega$ allows only finitely many 0-s in any accepted word. So a word like $(01)^\omega$ is not accepted, whereas this word is an element of $L3^\omega$.
2. $((a + b + c)^* b + c)^\omega$ is one possible expression specifying this language.
 3. The regular property specifying a single transaction given in Exercise A.15 is $\alpha^*(\varepsilon + m\alpha^*(d\alpha^*p + p\alpha^*d)\alpha^*)$ with $\alpha = s + a + b + h + t$. We may simply repeat that behavior infinitely often: $(\alpha^*(\varepsilon + m\alpha^*(d\alpha^*p + p\alpha^*d)\alpha^*))^\omega$ to obtain the specification. Note that it is possible that the coffee machine provides only finitely many (even 0) drinks followed by infinitely many aborted transactions. But this is proper behavior.

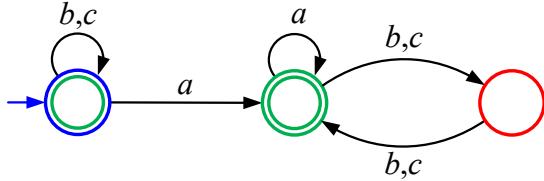
Exercise A.17 (ω -Regular expressions).

- For Example A.17, consider the automaton for the discussed language given in Figure A.3(a).



From this automaton, we can deduce an ω -regular expression following the pattern of ω -regular expressions discussed with Figure A.13. Expression $(a + b + c)^*a$ captures all finite words leading to the final state. Expression $a + c$ captures all words from the final state returning to the final state. Together this results in expression $(a + b + c)^*a(a + c)^\omega$ that captures the language.

- For Example A.18, consider the automaton given in Figure A.14.



From this automaton, we can deduce an ω -regular expression following the pattern of ω -regular expressions discussed with Figure A.13.

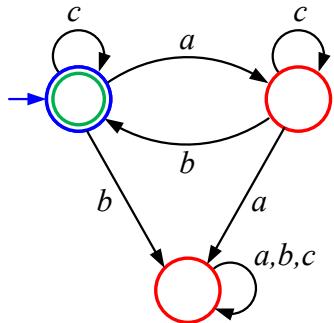
Expression $b + c$ captures the two options to go from the first final state to itself. No steps are needed to reach this final state. So this leads to a summand $(b + c)^\omega$ in the ω -regular expression we are looking for.

Expression $(b + c)^*a$ captures all finite words leading from the initial state to the second final state. Expression $a + (b + c)(b + c)$ captures all options from this final state to return to itself. This leads to summand $(b + c)^*a(a + (b + c)(b + c))^\omega$.

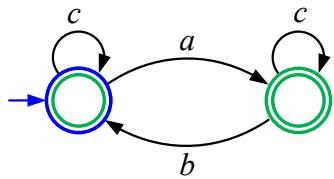
Together this results in expression $(b + c)^\omega + (b + c)^*a(a + (b + c)(b + c))^\omega$ that captures the language.

Exercise A.18 (Büchi automata).

- The automaton captures a request-response pattern. Every request (a) is always followed by a response (b). In between requests and responses, other behavior (c) may occur. Note that also the behavior without any request-response pair is ok. ω -regular expression: $(c + ac^*b)^\omega$.
- Yes, it is deterministic. For every combination of state and symbol, at most one transition with that symbol originates from that state.
- No, it is not complete. It is, for example, not specified what happens with a b in the initial state. The completed DBA is as follows:



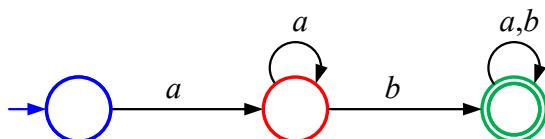
4. The automaton that we are considering is the following:



The language changes, in the sense that it now contains additional words with one more a than it has b -s. That is, the last request may never be followed by a response. The ω -regular expression that results from the pattern for ω -regular expressions is $(c + ac^*b)^\omega + c^*a(c + bc^*a)^\omega$. The second summand covers the extra words accepted by the automaton.

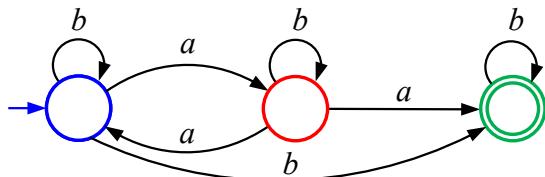
Exercise A.19 (ω -regular expressions, Büchi automata).

1. An ω -regular expression for this language is $aa^*b(a + b)^\omega$.
2. A DBA that accepts the language is the following.



Exercise A.20 (ω -regular expressions, Büchi automata).

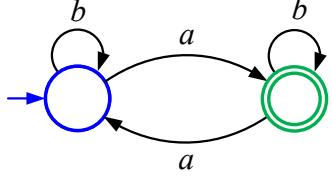
1. An NBA that accepts the language is the following.



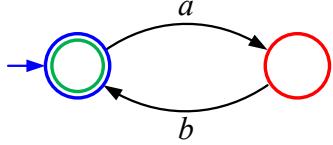
One has to take care that only words with finitely many pairs of a -s are accepted.

2. One possible ω -regular expression is $(b^*ab^*a)^*b^\omega$.

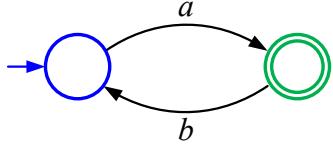
3. The following DBA accepts the complement language that contains all infinite words with infinitely many or an odd number of a -s.



Exercise A.21 (Finite automata, Büchi automata). The following two automata B_1 and B_2 provide a compact example:



and



Then $\mathcal{L}^\omega(B_1) = \mathcal{L}^\omega(B_2)$ specified by $(ab)^\omega$. But $\mathcal{L}(B_1)$ corresponds to $(ab)^*$ and $\mathcal{L}(B_2)$ corresponds to $a(ba)^*$.

Exercise A.22 (NB/FA, DB/FA, and limits).

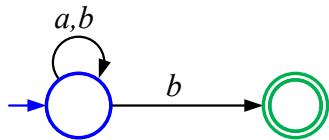
- We may deduce a ω -regular expressions from the given automata following the pattern of ω -regular expressions discussed with Figure A.13. For B , this yields: $b^*a((a+b)b^*a)^\omega$. A more compact alternative is $(b^*a)^\omega$. For C , this yields: $(a+b)^*bb^\omega = ab^\omega$, which can be simplified to $(a+b)^*b^\omega$.
- The regular expressions for $\mathcal{L}(C)$ and $\mathcal{L}(B)$ are as follows. For B , we have $b^*a((a+b)b^*a)^*$. (Note aside: Can this be simplified to $b^*a(b^*a)^*$?) For C , we have $(a+b)^*bb^* = ab^*$.

The equality $\mathcal{L}(C) = \overline{\mathcal{L}(B)}$ does not hold. B , interpreted as a DFA, and C , interpreted as an NFA, both do not accept words like aa , where two a -s occur in sequence. So the (finite-word) languages accepted by these two automata cannot be each other's complement.

- Since $\mathcal{L}^\omega((a+b)^*b^\omega)$ is the language accepted by C (see the first item of this exercise), and since we know from Example A.19 that this language is not deterministically Büchi recognizable, from Proposition A.2, we know that equality $\mathcal{L}^\omega((a+b)^*b^\omega) = \lim(\mathcal{L}(\alpha))$ cannot hold.

$\lim(\mathcal{L}(\alpha)) = \lim(\mathcal{L}((a+b)^*b))$ contains words with infinitely many a -s, because $(ab)^n$ for any $n \in \mathbb{N} \setminus \{0\}$, are elements of $\mathcal{L}(\alpha)$. $(ab)^\omega$ is therefore a word of $\lim(\mathcal{L}(\alpha))$. But it is not an element of $\mathcal{L}^\omega((a+b)^*b^\omega) = \mathcal{L}^\omega(C)$, distinguishing the two languages.

4. The correct NFA is the following:



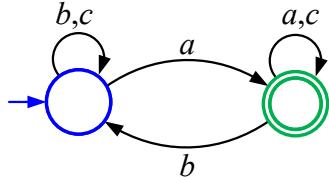
Since it has no cycle through a final state, it accepts ω -regular language \emptyset , defined by ω -regular expression \emptyset^ω .

(It may now be interesting to consider the smallest DFA for α and the ω -regular language accepted by that DFA.)

Exercise A.23 (NB/FA, DB/FA, and limits).

1. We may deduce ω -regular expressions from the given automata following the pattern of ω -regular expressions discussed with Figure A.13. For B , this yields $(b+c+a(a+c)^*b)^\omega$. For C , this yields $(a+b+c)^*a(a+c)^\omega$.
2. The regular expressions follow the expressions derived in the previous item. For B , we get $(b+c+a(a+c)^*b)^*$. For C , we have $(a+b+c)^*a(a+c)^*$.

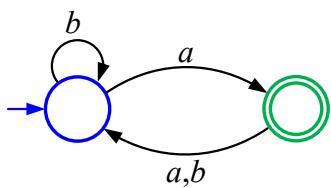
The equality $\mathcal{L}(C) = \overline{\mathcal{L}(B)}$ holds. $\overline{\mathcal{L}(B)} = \mathcal{L}(\bar{B})$, because B is deterministic and complete. \bar{B} gives the following automaton:



It can be seen that this accepts the same language as C . You may use the workbench to check this.

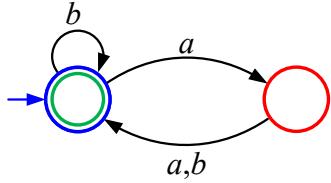
3. $\mathcal{L}^\omega(C)$ allows only words with finitely many b -s. $\text{lim}(\mathcal{L}(C))$ also allows words with infinitely many b -s. $\text{lim}(\mathcal{L}(C))$ contains words with infinitely many b -s, because $(ba)^n$ for any $n \in \mathbb{N} \setminus \{0\}$, are elements of $\mathcal{L}(C)$. $(ba)^\omega$ is therefore a word of $\text{lim}(\mathcal{L}(C))$. But since it is not a word of $\mathcal{L}^\omega(C)$, this example distinguishes the two languages.
4. (Above exam level) The reasoning follows the reasoning in Example A.20.

Exercise A.24 (Complement of Büchi automata?). Reconsider the first automaton given in Exercise A.22 as automaton A :



This is a complete DFA.

Its complement is the following:



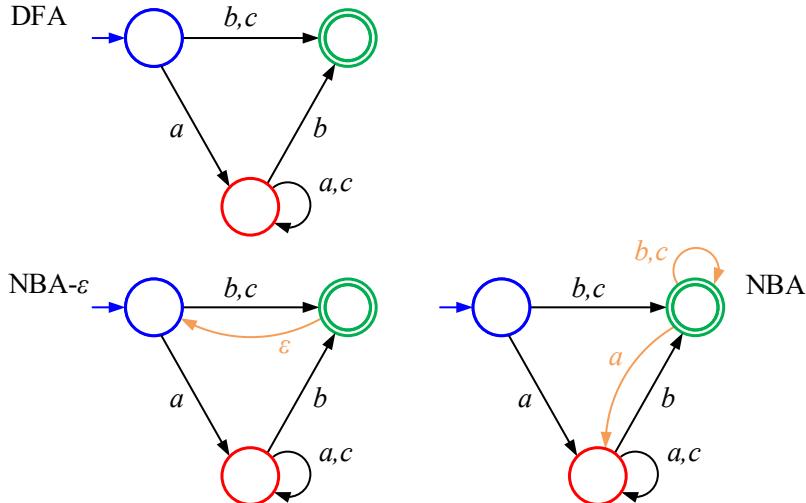
Considering both automata as Büchi automata, it can be seen that a^ω is accepted by both automata. Hence, $\mathcal{L}^\omega(A) \neq \overline{\mathcal{L}^\omega(\bar{A})}$.

Exercise A.25 (Non- ω -regular language). Consider the language $L9^\omega = \{(a^n b^n)^\omega \mid n \in \mathbb{N}\}$. The reasoning to show that this language is not regular follows the same line of reasoning as the reasoning in Example A.9 for (finite-word) regular languages showing that $L9 = \{a^n b^n \mid n \in \mathbb{N}\}$ is not regular.

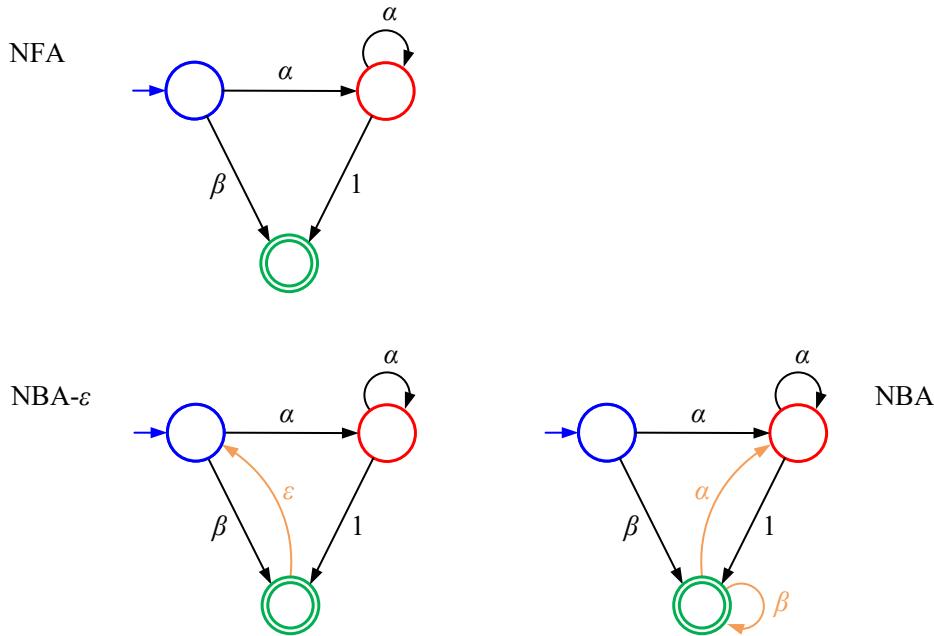
Assume towards a contradiction that an NBA B exists accepting $L9^\omega$. B must then distinguish all a^n subwords. Using the pigeonhole principle, this cannot be done with finitely many states. Hence B cannot exist, and hence $L9^\omega$ cannot be ω -regular.

Exercise A.26 (From ω -regular expression to NBA).

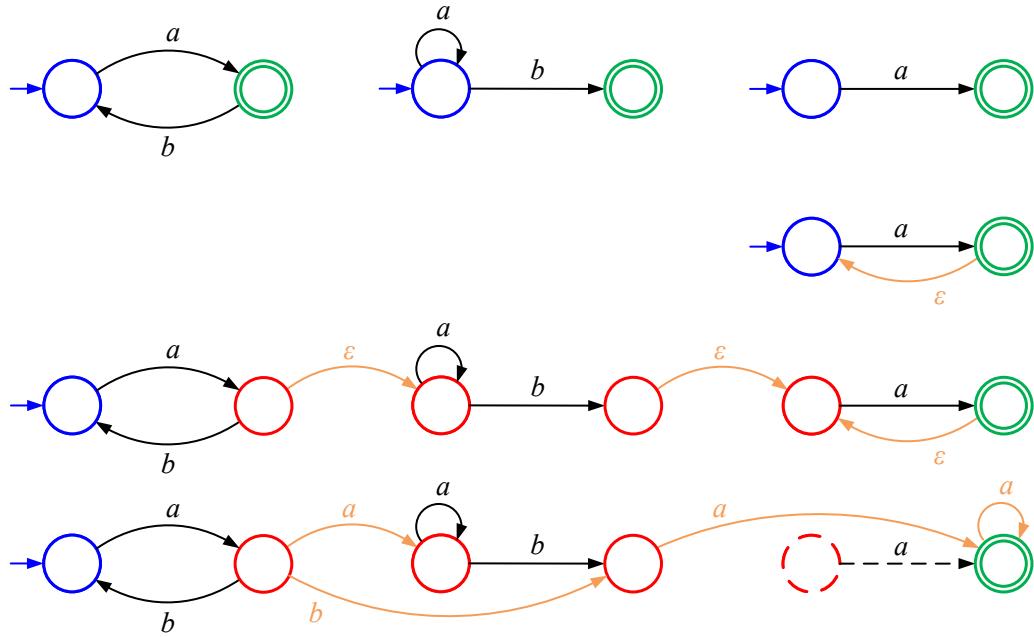
- The expression at hand is $(b + c + a \cdot (a + c)^* \cdot b)^\omega$. The following shows the conversion.



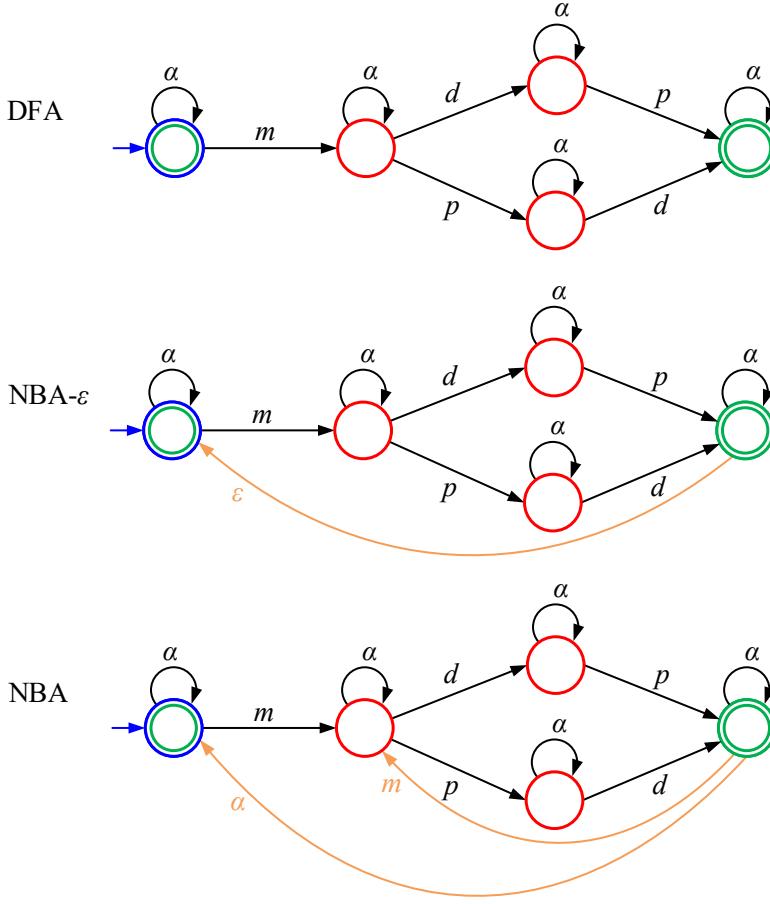
- The ω -regular expression at hand is $(\alpha^* \cdot 1 + \beta)^\omega$, with $\alpha = 0+1+2+3+4+5+6+7+8+9$ and $\beta = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$. The conversion goes as follows. Note that the conversion starts from a nondeterministic finite automaton (because α and β both allow symbols 1 through 9).



3. The ω -regular expression at hand is $\alpha\beta\gamma^\omega$ with $\alpha = a(ba)^*$, $\beta = a^*b$, and $\gamma = a$. The following shows the conversion. After ϵ elimination, the dashed state and transition are redundant.



4. The ω -regular expression at hand is $(\alpha^*(\epsilon + m\alpha^*(d\alpha^*p + p\alpha^*d)\alpha^*))^\omega$ with $\alpha = s + a + b + h + t$. The conversion is as follows. The ϵ move from the initial state to itself that results from the construction of the infinite concatenation is omitted.



Exercise A.27 (From NBA to ω -regular expression).

1. The NBA and the state-elimination process are visualized below. The NBA has two final states. We build the ω -regular expression from three finite-word regular expressions, describing the languages accepted by NFA A_{11} , A_{12} , and A_{22} , respectively. We use a shorthand notation for powers of a regular expression. The state elimination gives the following result:

$$\alpha_{11} = (b + c)^*$$

$$\alpha_{12} = (b + c)^*a(a + (b + c)^2)^*$$

$$\alpha_{22} = (a + (b + c)^2)^*$$

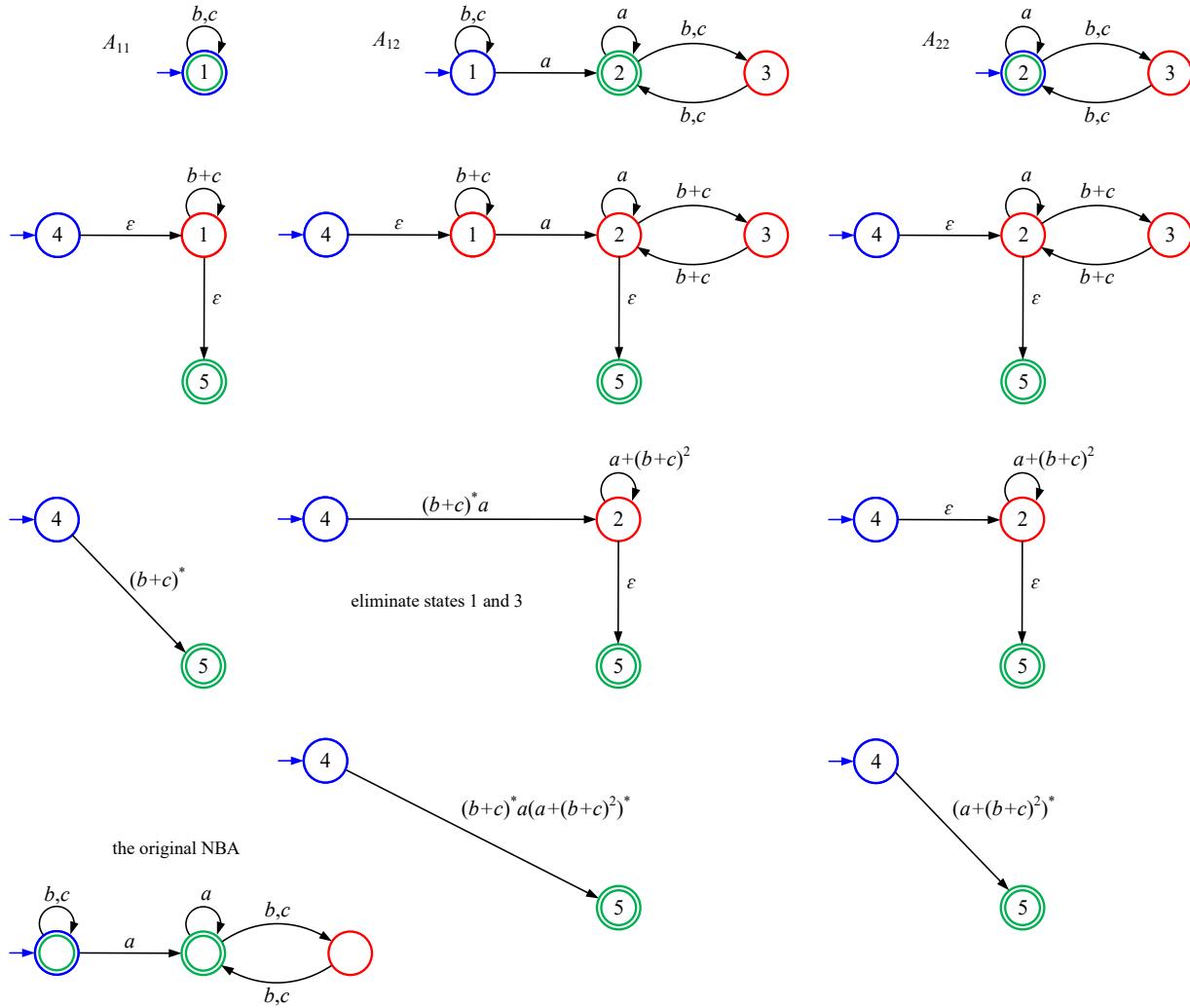
This leads to the following ω -regular expression:

$$\alpha_{11}^\omega + \alpha_{12}\alpha_{22}^\omega = ((b + c)^*)^\omega + (b + c)^*a(a + (b + c)^2)^*((a + (b + c)^2)^*)^\omega$$

This can be simplified by moving finite repetitions into the infinite repetitions:

$$(b + c)^\omega + (b + c)^*a(a + (b + c)^2)^\omega$$

This is the same expression as the one we saw in Exercise A.17.



2. The state elimination for the first NBA of Figure A.15 is shown in Figure A.23, showing the original NBA at the top. This leads to the following ω -regular expression:

$$a^*b(b + aa^*b)^*((b + aa^*b))^{\omega}$$

This can be simplified to $(a^*b)^{\omega}$, which is the same expression as given in Figure A.15.

The second NBA and the three DFA from which an ω -regular expression for this NBA can be built are shown in Figure A.24 (where we omitted redundant states from the DFA). This gives ω -regular expression $((ab)^{\omega} + a^*ba^*(a^*)^{\omega}$, which can again be simplified to the expression given in Figure A.15.

To convert the final NBA of Figure A.15 to an expression, we need finite-word regular expressions for the two NFA shown in Figure A.25. State elimination on those NFA and putting the result together gives the following ω -regular expression:

$$((ab)^*aa^*b + (ab)^*b)a^*(a^*)^{\omega}$$

As expected, also this result can be simplified to the expression given in Figure A.15.

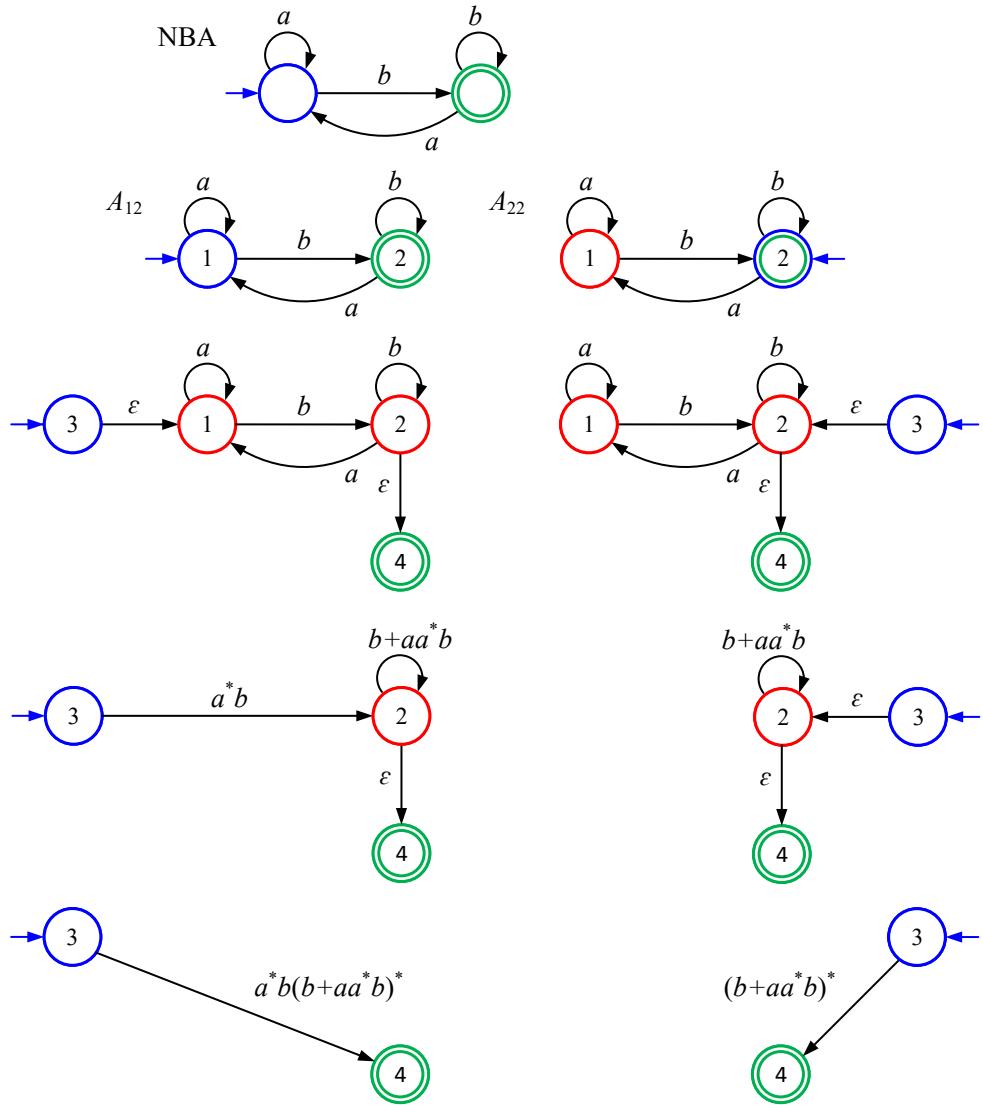


Figure A.23: Converting the first NBA of Figure A.15 to a regular expression via state elimination.

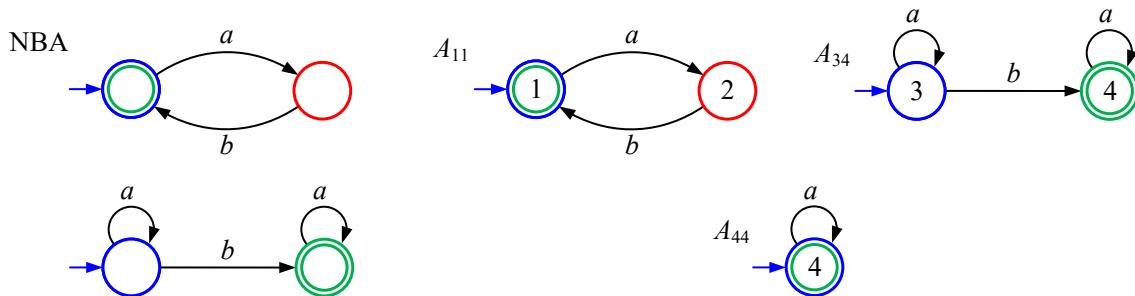


Figure A.24: Converting the second NBA of Figure A.15 to a regular expression.

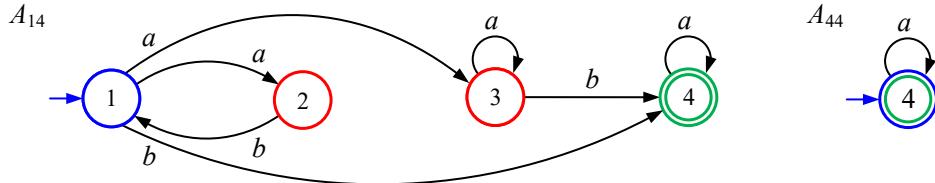
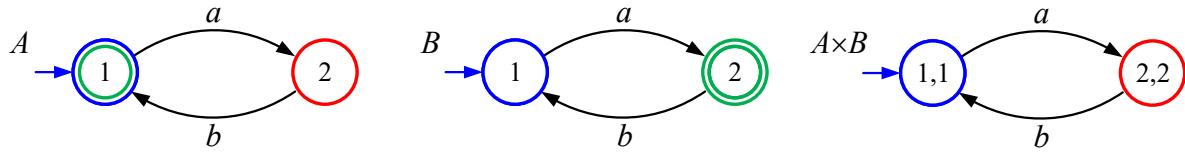


Figure A.25: Converting the third NBA of Figure A.15 to a regular expression.

Exercise A.28 (Product automata). The product according to Definition A.10 is as follows:



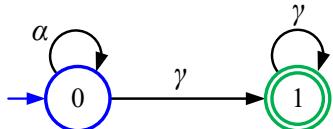
The product automaton has no final states. It therefore accepts the empty language \emptyset (which is both regular and ω -regular).

Since all (finite) words in $\mathcal{L}(A)$ end with a b and all words in $\mathcal{L}(B)$ with an a , their intersection is empty and it is correct that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B) = \emptyset$.

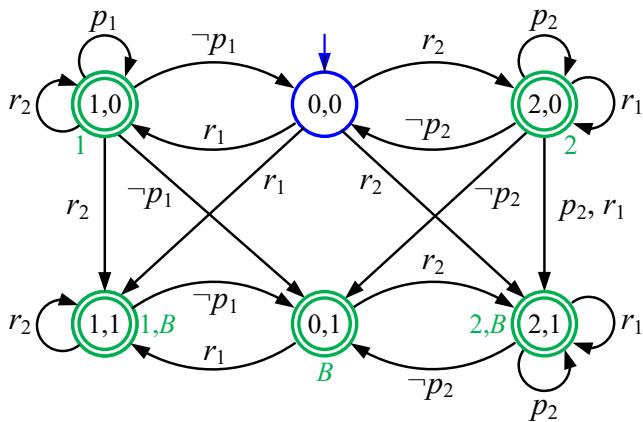
Since both A and B accept the ω -regular language $(ab)^\omega$, the intersection should also be this language. Hence, $\mathcal{L}^\omega(A) \cap \mathcal{L}^\omega(B) \neq \mathcal{L}^\omega(A \times B) = \emptyset$.

Exercise A.29 (Checking ω -regular properties).

1. A model for the bad behaviors according to the alternative property is the following, with $\alpha = r_1 + p_1 + \neg p_1 + r_2 + p_2 + \neg p_2$, as in Example A.25, and $\gamma = r_1 + \neg p_1 + r_2 + p_2 + \neg p_2$.



This two-state NBA accepts all words with an infinite tail in which no p_1 occurs. The product with the system S is the following GNBA:



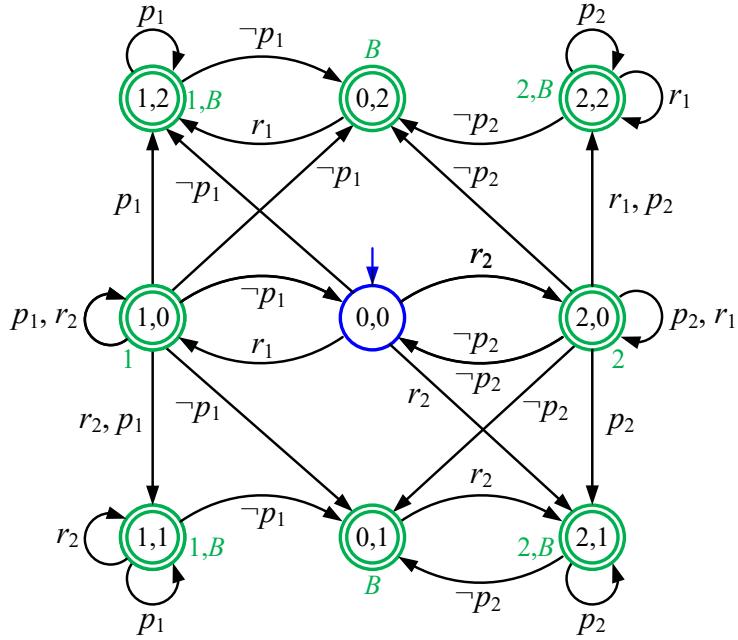
The language of this product is not empty. Within the $(i, 1)$ states, it is possible to visit all three final-state sets infinitely often. Thus, the system does not satisfy the

alternative property. $(r_1 \neg p_1 r_2 \neg p_2)^\omega$ has an accepting run in the product automaton, providing a counter example.

2. We may create an automaton for the bad behaviors using the observation that any behavior in which either one of the two processes does not receive access to the processor infinitely often is unacceptable. This can be captured in a three-state NBA, following the pattern seen in Example A.25. In the CMWB input format, this NBA can be specified as follows:

```
finite state automaton B {
    S0 initial -- r1,r2,p1,p2,np1,np2 --> S0
    S0 -- r2,p1,p2,np1,np2 --> S1 final
    S1 -- r2,p1,p2,np1,np2 --> S1
    S0 -- r1,p1,p2,np1,np2 --> S2 final
    S2 -- r1,p1,p2,np1,np2 --> S2
}
```

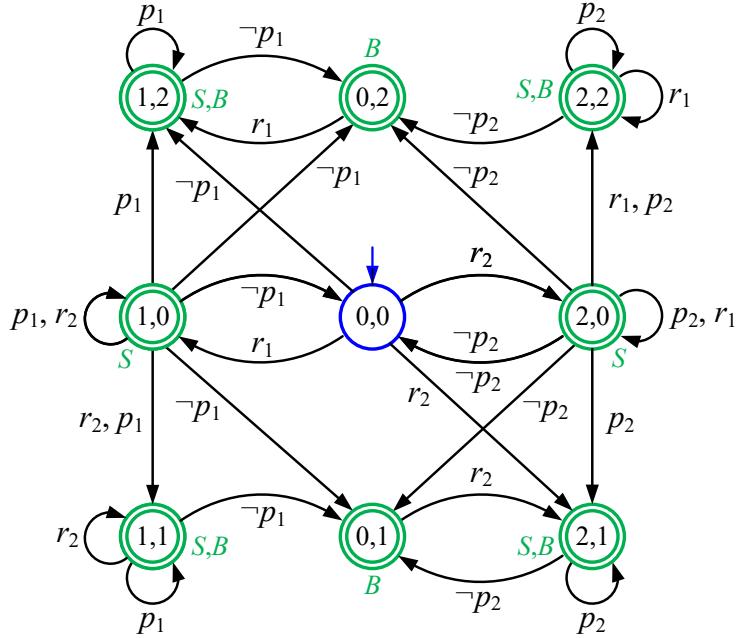
Taking the product of this NBA with the system S gives a nine-state GNBA with three final-state sets, in line with the product given in Example A.25:



Following the reasoning in that example, it is not difficult to argue that also this product has an empty language. Therefore, the system satisfies this property. You may verify your answer using the CMWB.

3. The product of the adapted system S with the bad behaviors specified in the previous item is a nine-state GNBA with only *two* final-state sets, as shown below. The adapted system model does not satisfy the expected behavior. Any word that visits any one of the (S, B) -labeled final states infinitely often is now acceptable, so, for instance, $(r_2 \neg p_2)^\omega$ is an unacceptable behavior present in the language of the product and hence

accepted by the adapted system model.



4. We need to extend the system model of the example with two additional states, one to keep track of whether or not P_2 may pre-empt P_1 , and one to ensure that P_2 is allowed to do a p_2 action after it pre-empted P_1 . This results in the following system model, using the CMWB input format:

```
finite state automaton S {
    S0 initial; final -- r1 --> S1 final
    S0 -- r2 --> S2 final
    S1 -- p1,r2 --> S3 final
    S1 -- np1 --> S0
    S3 -- p1 --> S3
    S3 -- np1 --> S0
    S3 -- r2 --> S4 final
    S2 -- p2 --> S2
    S2 -- r1 --> S1
    S2 -- np2 --> S0
    S4 -- p2 --> S2
}
```

The bad behaviors are those (infinite) words in which a pair of r_2 actions is not followed by a p_2 action. An NBA for the bad behaviors is the following:

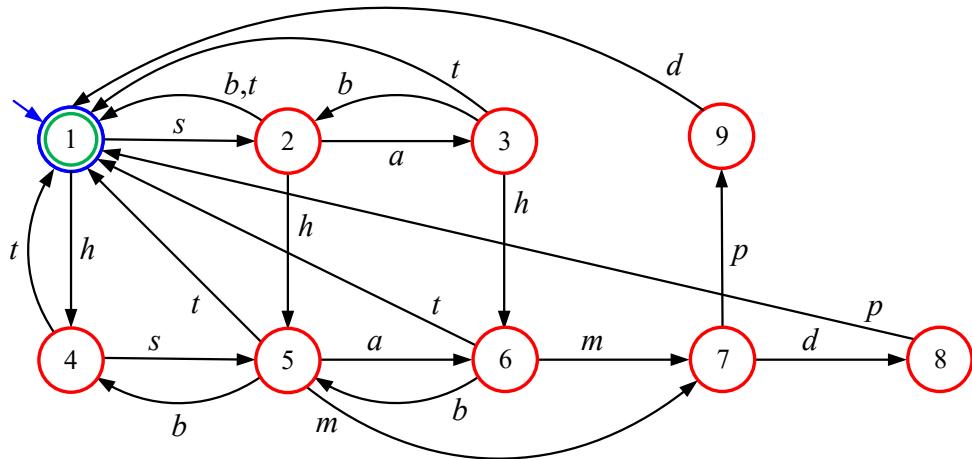
```
finite state automaton B {
    S0 initial -- r1,r2,p1,p2,np1,np2 --> S0
    S0 -- r2 --> S1
    S1 -- r2 --> S2
    S2 -- r1,r2,p1,np1,np2 --> S3 final
```

S3 -- r1,r2,p1,p2,np1,np2 ----> S3
}

The model satisfies the expected behavior. You may verify this using the CMWB or by manually constructing the product and checking this product for emptiness.

Exercises A.30, A.36, A.41 – Coffee machine.

Exercise A.30 (Checking ω -regular properties). We first adapt the model of the coffee machine to an NBA that repeatedly performs transactions. This can be done by merging the final and initial state of the DFA given in Example A.14.



The bad behaviors of the coffee machine are those in which at some point an m action is not followed by a combination of d and p in any order. We can specify those as an NBA, using the CMWB input format, as follows:

```

finite state automaton B {
    S0 initial -- s,a,b,h,t,m,d,p --> S0
    S0 -- m --> S1 final
    S1 -- s,a,b,h,t --> S1
    S1 -- d,p --> S2 final
    S2 -- s,a,b,h,t --> S2
}

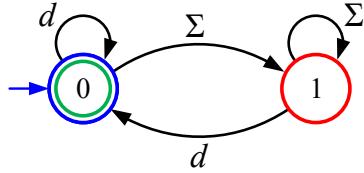
```

The idea behind this automaton is that, at some point, after doing an m , no d or p occur at all (state S1) or only one of those two actions occurs (state S2).

We may now perform the product between the NBA specifying the coffee machine and the NBA specifying the bad behaviors. Use the CMWB to complete the process of checking whether the coffee machine satisfies the desired behavior. The result should be that it does.

Exercise A.36 (Expansion). Let's first consider the property that the coffee machine always eventually serves a drink: $\square\Diamond d \equiv \Diamond d \wedge \square\Box\Diamond d \equiv (d \vee \square\Diamond d) \wedge \square\Box\Diamond d \equiv (d \wedge \square\Box\Diamond d) \vee (\square\Diamond d \wedge \square\Box\Diamond d) \equiv (d \wedge \square\Box\Diamond d) \vee \square(\Diamond d \wedge \Box\Diamond d) \equiv (d \wedge \square\Box\Diamond d) \vee \square(\Diamond d \wedge \Diamond d \wedge \Box\Diamond d) \equiv (d \wedge \square\Box\Diamond d) \vee \square(\Diamond d \wedge \square\Box\Diamond d) \equiv (d \wedge \square\Box\Diamond d) \vee \square(\Box\Diamond d) \equiv (d \wedge \square\Box\Diamond d) \vee (\text{true} \wedge \Box\Diamond d)$. So

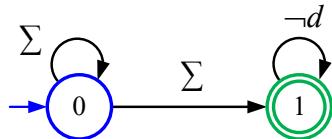
the final equality becomes $\square\Diamond d \equiv (d \wedge \square\Diamond d) \vee (\text{true} \wedge \square\Diamond d)$. This equality suggests an NBA with two states, one *final* state corresponding to formula $\square\Diamond d$ and one non-final state corresponding to this formula.



The result is very close to the DBA given in Example A.28. It only has extra d transitions from state 0 to state 1 and from state 1 to itself, turning those into Σ transitions when combined in shorthand notation with the already present $\neg d$ transitions in Example A.28. This extra nondeterminism does not change the language accepted by the automaton. The Σ -transitions correspond to the **true** conjunct in the expression resulting from the expansion.

Considering the second property, let $\phi = ((\neg m) \cup d) \wedge ((\neg m) \cup p)$ and $\psi = m \rightarrow \Diamond\phi$. The property of interest then becomes $\square\psi$. This can be rewritten to $(\neg m \vee \Diamond\phi) \wedge \Diamond\psi \equiv (\neg m \wedge \Diamond\psi) \vee (\phi \wedge \Diamond\psi) \equiv (\neg m \wedge \Diamond\psi) \vee (\text{true} \wedge \Diamond(\phi \wedge \Diamond\psi))$. This result suggests the initial $\neg m$ self-loop we see in Example A.28. The expansion can be continued, but the process is tedious. As a further exercise, one could try to expand one of the until subformulae.

Exercise A.41 (Checking properties). Again first consider the property that the coffee machine always eventually serves a drink: $\square\Diamond d$. The bad behaviors are then $\neg\square\Diamond d$. This can be rewritten to $\Diamond\square\neg d$. This gives the following NBA B :



At some point, with or after the last d , the automaton nondeterministically jumps to the final state, where d actions are no longer possible. Although it may seem obvious that the property should be satisfied, this is not the case. The product is sketched in Figure A.26. It uses the NBA model of the coffee machine given in the answer to Exercise A.30.

Intuitively, the product has two copies of the states of the coffee machine, one for each state of the NBA B . All transitions of the first copy are preserved. This corresponds to the Σ -labeled transition from state 0 to itself in B . It is possible to nondeterministically jump to the second copy with any action. The second copy allows all actions except d . It is not possible to return to the first copy. The product has two final-state sets, corresponding to the final states of the machine model S and the bad behaviors B .

The language of this product is not empty. For example, the word $(st)^\omega$ is accepted. There is a run that goes infinitely often through state $(1,1)$, which belongs to both final-state sets. This run specifies the behavior that repetitively a drink is selected after which the machine times out and returns to its initial state. This is allowed behavior, so the property is correctly rejected.

For the second property, it may be the easiest to directly derive an NBA for the bad behaviors. The following could be such an NBA:

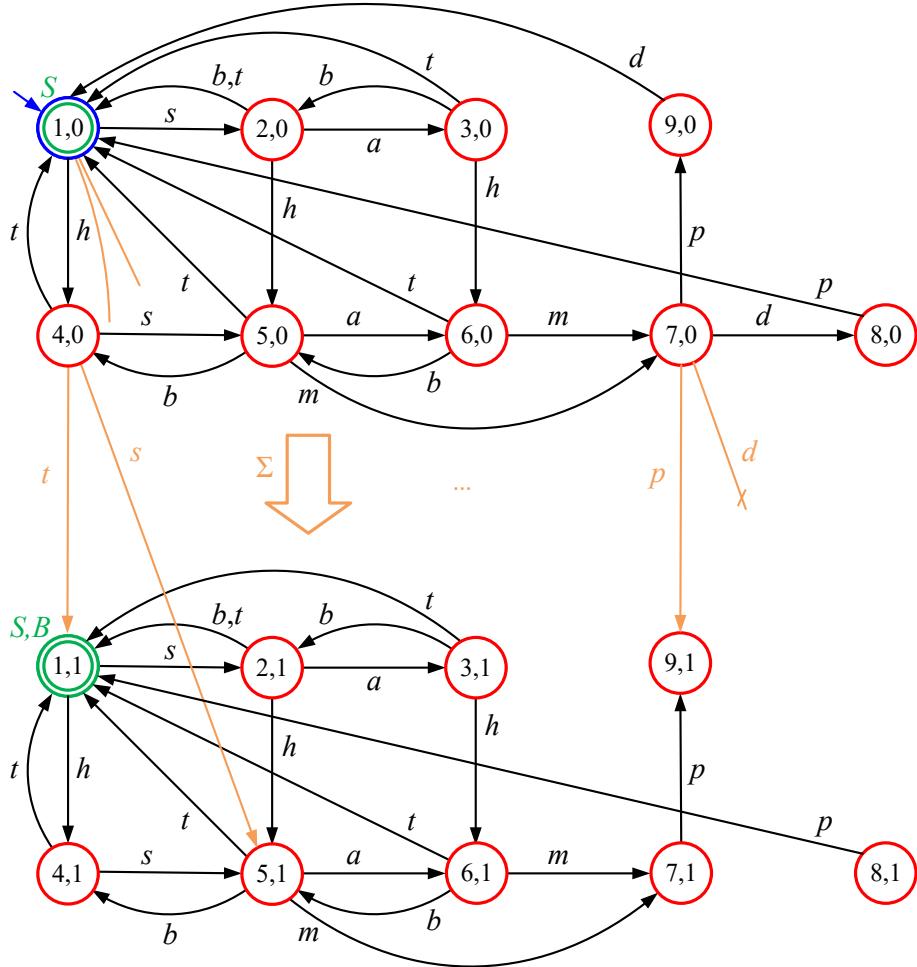
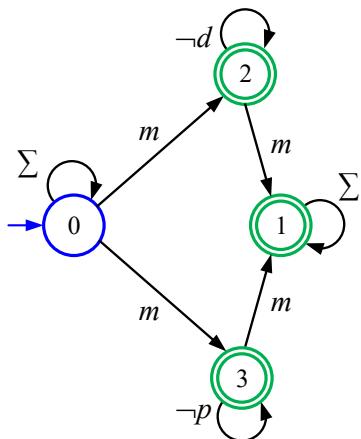


Figure A.26: The product GNBA for verifying whether the coffee machine continuously serves drinks.



This NBA states that at some point an m has to occur, which is then *not* followed by either a d or a p , or which *is* followed by a second m before completing the transaction. It is interesting to compare the NBA with the NBA for the bad behaviors given as an answer

in Exercise A.30.

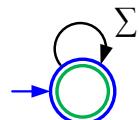
For illustration purposes, we consider the expansion of the LTL formula as well. Recall that we introduced $\phi = ((\neg m) \cup d) \wedge ((\neg m) \cup p)$ and $\psi = m \rightarrow \diamond \phi$. The property of interest then is $\square \psi$. The bad behaviors are now $\neg \square \psi \equiv \diamond \neg \psi$. Then, $\neg \psi = \neg(m \rightarrow \diamond \phi) \equiv \neg(\neg m \vee \diamond \phi) \equiv (m \wedge \neg \diamond \phi)$. Next, $\neg \phi = \neg(((\neg m) \cup d) \wedge ((\neg m) \cup p)) \equiv \neg((\neg m) \cup d) \vee \neg((\neg m) \cup p)$. We continue by expanding one of the until subformulae as follows: $\neg((\neg m) \cup d) \equiv \neg(d \vee (\neg m \wedge \diamond((\neg m) \cup d))) \equiv \neg d \wedge (m \vee \diamond((\neg m) \cup d))$. Putting everything together yields $\diamond(m \wedge \diamond(\neg d \wedge (m \vee \diamond((\neg m) \cup d)))) \vee (\neg p \wedge (m \vee \diamond((\neg m) \cup p))))$. The practiced reader may recognize the structure of the above NBA also in the resulting LTL formula.

Creating a product manually is challenging. You may use the CMWB instead. As we saw earlier, the property is satisfied. It should not be difficult to see from the coffee-machine model (given in the answer to Exercise A.30) that, once an m has occurred, both a d and a p will follow, while a second m cannot occur until the transaction has completed.

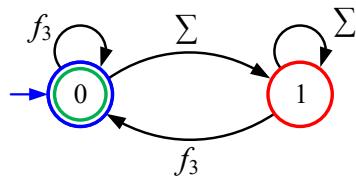
Exercises A.31, A.39, A.43 – Arbiter.

Exercise A.31 (Propositional formula). Recall the alphabet Σ defined in Example A.26 (Arbiter). Further recall the atomic propositions $grnt_i$ and ack_i defined in that example. The service-provision property can then be defined as $(req_1 \vee req_2 \vee req_3) \rightarrow (grnt_1 \vee grnt_2 \vee grnt_3)$.

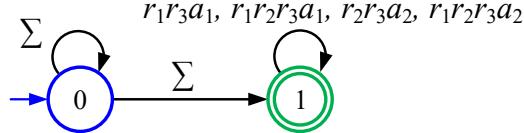
Exercise A.39 (Expansion, NBA). All of the symbols in the alphabet Σ defined in Example A.26 satisfy the propositional-logic formulae occurring in the mutual-exclusion property $(\neg(grnt_1 \wedge grnt_2) \wedge \neg(grnt_1 \wedge grnt_3) \wedge \neg(grnt_2 \wedge grnt_3))$, the no-waste property, $(grnt_1 \rightarrow req_1 \wedge grnt_2 \rightarrow req_2 \wedge grnt_3 \rightarrow req_3)$, and the service-provision property $((req_1 \vee req_2 \vee req_3) \rightarrow (grnt_1 \vee grnt_2 \vee grnt_3))$. This means that the three properties are all equivalent to $\square \text{true}$. This can be expanded to $\text{true} \wedge \square \text{true}$, which leads to the following NBA.



For the fairness property, consider, as an example, the one corresponding to Processor 3: $\square \diamond (grnt_3 \vee \neg req_3)$. The set of symbols from Σ that satisfy $grnt_3 \vee \neg req_3$ is the set $\{n, r_1a_1, r_1r_2a_1, r_2a_2, r_1r_2a_2, r_3a_3, r_1r_3a_3, r_2r_3a_3, r_1r_2r_3a_3\}$. Let's refer to this set as atomic proposition f_3 . The fairness property is then equivalent to $\square \diamond f_3$. The reasoning then follows the development for Exercise A.36, leading to the following NBA:



Exercise A.43 (Checking properties). The bad behaviors for the fairness property for Processor 3 are captured by $\neg \square \diamond f_3 \equiv \diamond \square \neg f_3$ with $\neg f_3 \equiv \{r_1r_3a_1, r_1r_2r_3a_1, r_2r_3a_2, r_1r_2r_3a_2\}$. An NBA B specifying the language is the following:



Taking the product with the arbiter NBA S given in Example A.26 gives a GNBA, illustrated in Figure A.27. The product is very similar to the product constructed for the coffee machine earlier. It essentially has two copies of the arbiter states, one corresponding to the 0 state of the property NBA and one to the 1 state of this NBA. It is possible to go from copy 0 to copy 1 with any action that can also be performed within the 0 copy. It is not possible to go back. The GNBA has two final-state sets. The language of the product is not empty. It accepts, for example, word $r_1r_3a_1(r_2r_3a_2\ r_1r_3a_1)^\omega$, which corresponds to the behavior where Processors 1 and 2 alternate their requests, thus starving Processor 3. The arbiter is therefore not fair.

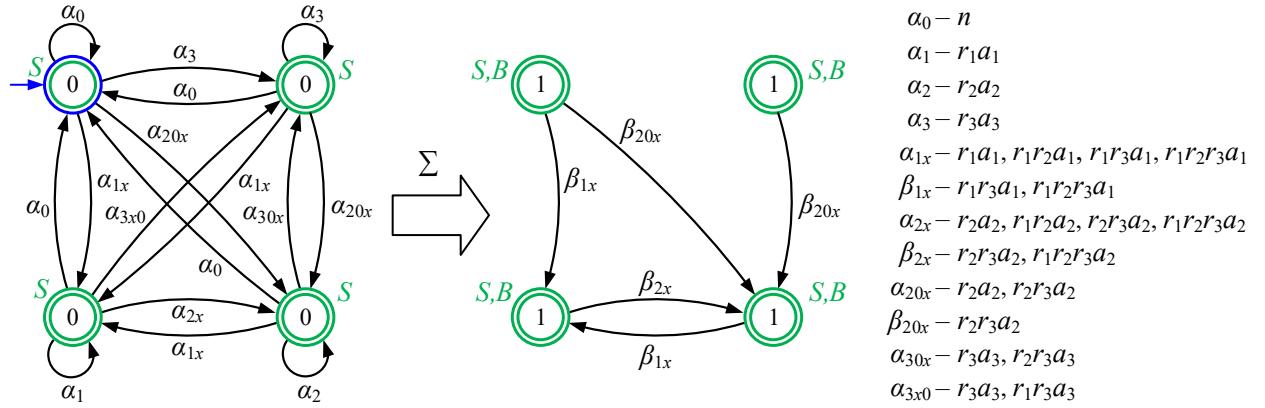


Figure A.27: The product GNBA for verifying whether the arbiter is fair.

The arbiter satisfies fairness for the other two processors, but that does not change the conclusion that the arbiter is not fair.

The arbiter can be made fair by introducing an extra state that gives Processor 3 the highest priority if Processors 1 and 2 get subsequent requests granted. The following shows a DBA in the workbench input format.

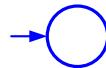
```
finite state automaton arbiter {
    s0 initial -- n ----> s0 final // alpha_0
    s0 -- r1a1,r1r3a1,r1r2a1,r1r2r3a1 --> s1 // alpha_1x
    s0 -- r2a2,r2r3a2 --> s2 // alpha_20x
    s0 -- r3a3 --> s3 // alpha_3
    s1 -- n --> s0 // alpha_0
    s1 -- r1a1 --> s1 final // alpha_1
    s1 -- r2a2,r1r2a2 --> s2 // alpha_2x
    s1 -- r2r3a2,r1r2r3a2 --> s5 final
    s1 -- r3a3,r1r3a3 --> s3 // alpha_3x0
    s2 -- n --> s0 // alpha_0
    s2 -- r1a1,r1r3a1,r1r2a1,r1r2r3a1 --> s1 // alpha_1x
```

```

s2 -- r2a2 --> s2 final // alpha_2
s2 -- r3a3,r2r3a3 --> s3 // alpha_30x
s3 -- n --> s0 // alpha_0
s3 -- r1a1,r1r3a1,r1r2a1,r1r2r3a1 --> s1 // alpha_1x
s3 -- r2a2,r2r3a2 --> s2 // alpha_20x
s3 -- r3a3 --> s3 final // alpha_3
s5 -- n --> s0
s5 -- r3a3,r1r3a3,r2r3a3,r1r2r3a3 --> s3
s5 -- r1a1,r1r2a1 --> s1
s5 -- r2a2 --> s2
}

```

It can be verified that this corrected model satisfies all four properties. The bad behaviors for mutual exclusion, no waste, and service provision are in all three cases the empty language: $\neg \Box \text{true} \equiv \Diamond \text{false}$. That is, an NBA for these properties is the following:



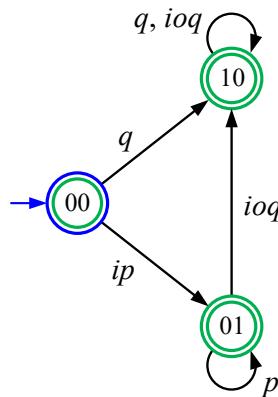
As a result, also the product automaton has no transitions and thus defines the empty language. This shows that the three properties are satisfied.

For the fairness property, we can re-use the above NBA B . The workbench can be used to verify that the product with the new NBA model for the arbiter gives indeed the empty language, showing that the corrected arbiter is fair.

Exercises A.32, A.34, A.37, A.40 – Analyzing a Mealy model.

Exercise A.32 (DBA system model, propositional specification). Following the arbiter example, we first build a set of symbols from the positive signals that may occur in the Mealy machine. The Mealy machine has four signals, being i, o, q , and p . We define symbol alphabet Σ as $\{n, i, o, q, p, io, iq, ip, oq, op, qp, ioq, iop, iqp, oqp, ioqp\}$, where n represents the case that no signal is positive.

1. Using the introduced symbols, we can define the following NBA, where states and transitions correspond one-to-one to the Mealy model. All states are accepting, because a Mealy machine does not have control over its inputs, meaning that all possible behaviors are acceptable system behaviors.

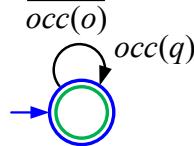


2. From the symbols, we define atomic propositions that coincide with the occurrence of an o or a q : $\text{occ}(o) = \{s \in \Sigma \mid o \in s\} = \{o, io, oq, op, ioq, iop, oqp, ioqp\}$ and $\text{occ}(q) = \{s \in \Sigma \mid q \in s\} = \{q, iq, oq, qp, ioq, iqp, oqp, ioqp\}$. The desired property can then be specified as $\text{occ}(o) \rightarrow \text{occ}(q)$.

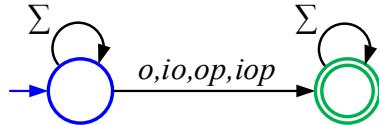
Exercise A.34 (LTL property specification). The desired property is a safety property: $\square(\text{occ}(o) \rightarrow \text{occ}(q))$.

Exercise A.37 (Expansion, NBA). The desired result for expansion directly follows from the expansion law for the always operator: $\square(\text{occ}(o) \rightarrow \text{occ}(q)) \equiv (\text{occ}(o) \rightarrow \text{occ}(q)) \wedge \square\square(\text{occ}(o) \rightarrow \text{occ}(q))$.

To come to an NBA, observe that $\text{occ}(o) \rightarrow \text{occ}(q) \equiv \neg\text{occ}(o) \vee \text{occ}(q)$. In turn, $\neg\text{occ}(o) \equiv \overline{\text{occ}(o)} = \Sigma \setminus \text{occ}(o)$. Note that $\overline{\text{occ}(o)}$ is an atomic proposition and hence a set of symbols. So the simplest possible NBA is then the following, with a single state and a self-loop with any symbol from $\text{occ}(o)$ or $\text{occ}(q)$. Informally, the state corresponds to the starting formula $\square(\text{occ}(o) \rightarrow \text{occ}(q))$ that re-occurs in the right-hand side of the above expansion; the transitions correspond to the two or-clauses resulting from rewriting the implication.

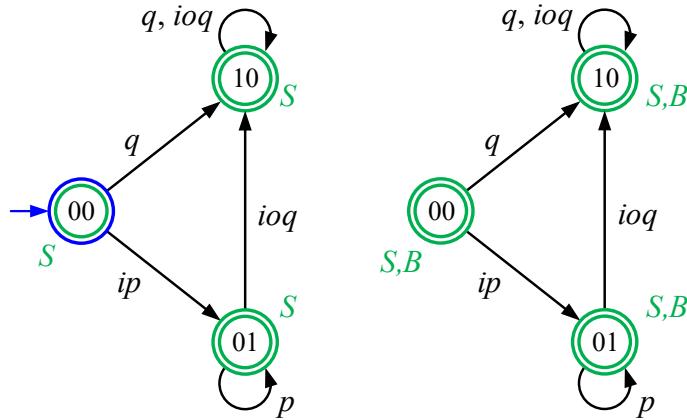


Exercise A.40 (Property checking). The bad behaviors are captured by the property $\neg\square(\text{occ}(o) \rightarrow \text{occ}(q)) \equiv \Diamond\neg(\neg\text{occ}(o) \vee \text{occ}(q)) \equiv \Diamond(\text{occ}(o) \wedge \neg\text{occ}(q))$. In turn, $\text{occ}(o) \wedge \neg\text{occ}(q)$ is equivalent to atomic proposition $\{o, io, op, iop\}$. Thus, the bad behaviors are specified by $\phi_B = \Diamond\{o, io, op, iop\}$. The language defined by this formula is captured by the following NBA B :



This NBA captures the fact that at any one point in the execution of the Mealy machine an o occurs that not also sets q . Before and after that event, any behavior (i.e., any symbol) is possible.

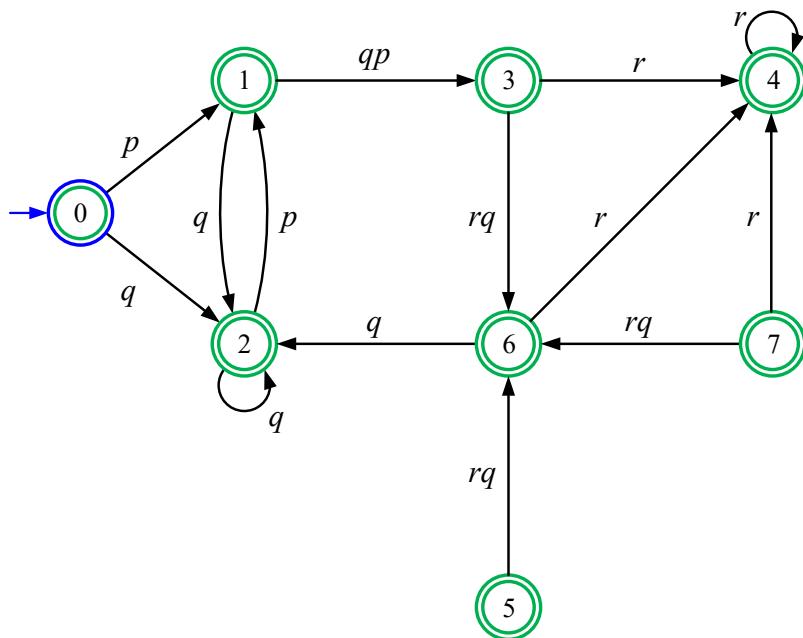
Referring to the model of Exercise A.32 as S , this leads to the product $S \otimes B$ shown below. The product has six states, with two final-state sets, labeled S and B , corresponding to the final state-sets of S and B . It is not possible though to go from any of the initial states to the final-state set labeled B . This means that the language accepted by the product is the empty language. This in turn means that the property is satisfied.



Exercises A.33, A.35, A.38, A.42 – Controller.

Exercise A.33 (DBA system model). We first need to build a set of symbols from the *relevant* signals that may occur in the Mealy machine. Only the state bits r , q , and p are relevant for the mentioned properties. We thus define Σ as $\{n, r, q, p, rq, rp, qp, rqp\}$, where n represents the case that none of the three signals is positive.

1. It is clear from the Mealy diagram that the first property is true. From the description, it further follows that mutual exclusion is only violated in states 5 and 7. Since both these states are not reachable from the initial state, also this property holds.
2. The following NBA can be used to verify the properties. The NBA states mimic the states of the Mealy model; the transitions abstract from the input and output values, corresponding to the encoding of the reached state instead. As before, all states are accepting. Note that states 5 and 7 could be dropped from the model, because they clearly do not have any impact on the acceptance of words.

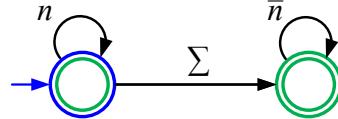


Exercise A.35 (LTL property specification). Both properties are safety properties.

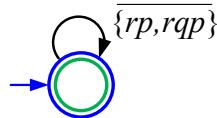
A transition to the initial state is captured by atomic proposition $\overline{\{n\}} = \Sigma \setminus \{n\}$. So the property that this state cannot be reached from any of the others is specified as $\square(\overline{\{n\}} \rightarrow \circlearrowleft \overline{\{n\}})$. So only if the machine happens to be in the initial state, then it can go there again, but once it has left the initial state, it cannot return.

Mutual exclusion at any given point in time corresponds to the atomic proposition $\overline{\{rp, rqp\}}$. So the mutual exclusion property can be specified as $\square \overline{\{rp, rqp\}}$.

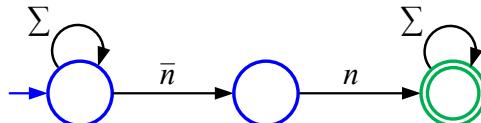
Exercise A.38 (Expansion, NBA). Let $\bar{n} = \overline{\{n\}}$ and $\phi = \bar{n} \rightarrow \circlearrowleft \bar{n}$. We moreover use n to refer to $\{n\}$. For the first property, which can now be written as $\square \phi$, the desired result then follows from the expansion law for the always operator followed by some further manipulation of the resulting formula: $\square \phi \equiv \phi \wedge \square \phi \equiv (\neg \bar{n} \vee \circlearrowleft \bar{n}) \wedge \square \phi \equiv (n \vee \circlearrowleft \bar{n}) \wedge \square \phi \equiv (n \wedge \square \phi) \vee (\circlearrowleft \bar{n} \wedge \square \phi) \equiv (n \wedge \square \phi) \vee \circlearrowleft (\bar{n} \wedge \square \phi) \equiv (n \wedge \square \phi) \vee (\text{true} \wedge \circlearrowleft (\bar{n} \wedge \square \phi))$. This last formula specifies two options in the expanded form consisting of a non-temporal property for the first symbol in an accepted word and a temporal part for the remainder of the word. The constant `true` indicates that any symbol in Σ is allowed as first symbol for the second option. Any new subformulae in the result could be further expanded till the point that no new formulae occur. The obtained subformulae then correspond to states in an NBA accepting the same language as the original formula. It is an interesting exercise to further pursue this expansion. The NBA that results and that accepts the same language as the formula $\square \phi$ is then the following:



Recall that $\overline{\{rp, rqp\}}$ is an atomic proposition and a set of symbols. The second property is a straightforward application of the expansion law for the always operator: $\square \overline{\{rp, rqp\}} \equiv \overline{\{rp, rqp\}} \wedge \square \overline{\{rp, rqp\}}$, which leads to the following automaton:



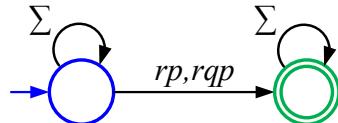
Exercise A.42 (Checking properties). First, consider the first property. The bad behaviors for this property are captured by the formula $\neg \square(\bar{n} \rightarrow \circlearrowleft \bar{n}) \equiv \diamond \neg(\neg \bar{n} \vee \circlearrowleft \bar{n}) \equiv \diamond(\bar{n} \wedge \neg \circlearrowleft \bar{n}) \equiv \diamond(\bar{n} \wedge \circlearrowleft n)$. This formula can be expanded, but also without expansion, it should be straightforward to verify that the language defined by this formula is captured by the following NBA *B1*:



This NBA captures the fact that at any one point in the execution of the Mealy machine a $\bar{n}n$ combination occurs, which is violating the fact that the initial state should not be reachable from any other state.

The next step is to take the product with the model S of Exercise A.33. This product has 24 states (when not applying any optimization), with two final-state sets, corresponding to S and $B1$. Also without constructing the product explicitly, it is clear that it is not possible to reach the $B1$ final-state set, because S does not have any n actions. This means that the language accepted by the product automaton is the empty language, confirming the conclusion already drawn in Exercise A.33 that the property is satisfied.

Second, consider the second property, $\square \overline{\{rp, rqp\}}$. This can be rewritten to $\diamond \neg \overline{\{rp, rqp\}} \equiv \diamond \{rp, rqp\}$, which states that eventually either rp or rqp occurs, violating mutual exclusion. This corresponds to the following NBA.



Again, because model S of Exercise A.33 has no rp or rqp actions, it is not difficult to see that the product language is empty. Hence, also this second property is satisfied.

B

Markov Modeling and Discrete-Event Simulation

Introduction

Part B of these course notes is concerned with the modeling and analysis of the evolution over time of systems or phenomena with a random character. Analysis examples include the expected time that a mars rover requires to explore a rocky landscape on mars, the probability of packet loss due to contention on a shared communication medium, or the expected amount of cash a gambler carries after a number of spins of the roulette wheel at the casino. As a modeling foundation we will use Markov chains, named after the influential nineteenth century Russian mathematician Andrei Andreyevich Markov shown in Figure B.1. Markov chains are sufficiently rich to model a wide class of systems, while at the same time have sufficient structure to allow the analysis of interesting stochastic properties. For this reason Markov chains are used in a very broad range of applications, including performance analysis of embedded systems, computer networks and cyber-physical systems, stochastic software testing, data compression, page ranking for websites, insurance risk management, and biomedical engineering.

We will review important results from discrete-time Markov chain theory and its application. Classical books on Markov chains are firmly anchored in probability theory. These course notes are self-contained and do not require such prior knowledge. We made an attempt to avoid the use of probability concepts as much as possible and will explain all the necessary concepts in the text. We refer to [16] for an excellent thorough treatment of probabilities. A classical introduction in stochastic processes (including Markov chains) is [7]. A modern introductory book including many examples is [23]. For a mathematical treatment of Markov theory, including the proofs of important theorems, we refer to [21, 6].

Learning objectives. After studying these notes, the student will be able to

- **understand** the concept of a discrete-time stochastic process;
- **understand** the concepts of variable dependence and variable distribution and



Figure B.1: Andrei Andreyevich Markov (source: Wikipedia)

argue whether variables are dependent/independent and identically distributed or not;

- **read, understand, design and formally describe** Markov chains and reward models;
- **understand and compute** the transient distributions and n-step probabilities for any Markov chain (both in terms of probability matrices and in terms of paths);
- **understand and compute** the transient expected rewards for any Markov reward model;
- **understand and determine** the communicating states and classes, recurrent/transient states and classes and periodic/aperiodic states and classes of any Markov chain;
- **know** the concepts of irreducible Markov chain, ergodic Markov chain, ergodic class and unichain;
- **understand and compute** the (return) probabilities to hit a state or set of states for any Markov chain (both in terms of linear equations and in terms of paths);
- **understand and compute** the expected cumulative reward earned until hitting a state or set of states for any Markov reward model (both in terms of linear equations and in terms of paths);
- **reason** when (Cezàro) limiting distributions and limiting probability matrices exist and **compute** them (based on balance equations and hitting probabilities) for any Markov chain;

- **reason** when long-run expected rewards or long-run expected average rewards exist and **compute** them for any Markov reward model;
- **understand** the strong law of large numbers and the central limit theorem for processes of independent identically distributed variables and **derive** confidence interval and (absolute/relative) errors in the estimation by discrete-event simulation of expected cumulative rewards and long-run expected (average) rewards.

B.1 Discrete-time stochastic processes

Consider a system that is observed at times $0, 1, \dots$. The random *state* of the system at time n ($n = 0, 1, \dots$) is denoted by X_n , where X_0 is called the *initial state*. Here n can refer to an actual moment in time, but can instead denote an index in a sequence of events. X_n assumes values in a finite set \mathcal{S} of states called the *state-space* of the system. For theoretical reasons (that will become clear later) we assume it to be of the form $\{1, 2, \dots, N\}$ for some natural number $N \geq 1$, but in practice we can choose any finite set of states. X_n is a *random variable* implying that it assumes values in \mathcal{S} with a certain *probability*.

$$\text{The probability that the system is in state } i \ (i \in \mathcal{S}) \text{ at time } n \ (n = 0, 1, \dots) \text{ is denoted by } P(X_n = i). \quad (\text{B.1})$$

The value i is called a *realization* of X_n , i.e. one of the values that X_n can take. A probability is represented by a real number in interval $[0, 1]$ and thus $0 \leq P(X_n = i) \leq 1$. At time n the system must always be in some state and therefore $\sum_{i \in \mathcal{S}} P(X_n = i) = 1$.

B.1.1 Probability distributions

The probabilities of X_n taking any of the states in \mathcal{S} is represented by the *probability distribution*¹ $\pi^{(n)} : \mathcal{S} \rightarrow [0, 1]$ defined by $\pi^{(n)}(i) = P(X_n = i)$ for all $i \in \mathcal{S}$. Because of the shape of \mathcal{S} we will often assume $\pi^{(n)}$ to be a row vector of the form $[\pi_1^{(n)}, \pi_2^{(n)}, \dots, \pi_N^{(n)}]$ where $\pi_i^{(n)}$ is the i^{th} element of $\pi^{(n)}$, so

$$\pi_i^{(n)} = P(X_n = i) \quad (\text{B.2})$$

The sequence of random variables X_0, X_1, \dots is called a *discrete-time stochastic process*. Since the state-space is discrete, the sequence is said to be *discrete-valued*. Discrete-time stochastic processes are used to model the evolution over time of systems or phenomena with a random character. For example X_n can model the number of packets in a buffer at time n , the waiting time of the n^{th} person arriving in a queue, or the amount of cash a gambler carries after n spins of a roulette wheel in the cassino.

¹In fact *probability mass function* would be a more appropriate term, but we will follow the convention in literature.

Example B.1 (Game of dice). As a simple example consider a game of dice in which a single player repeatedly tosses a die. We are interested in the probability of the player throwing a six after 10 tosses. We can model this game as a discrete-time stochastic process X_0, X_1, \dots with state-space $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$, where the states correspond to the number of dots on the six sides of the dice. The random outcome of the n^{th} toss is given by X_{n-1} (for $n \geq 1$). Since the outcomes of the different tosses are independent the probability distribution $\pi^{(n)}$ is given by $\pi^{(n)} = [\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}]$, at least if the player is dealing with a fair die. Therefore the probability of throwing a six after ten tosses is given by $P(X_9 = 6) = \frac{1}{6}$.

B.1.2 Rewards and expected rewards

Often we are not interested in the state of the system per se, but in the value that a certain (real-valued) variable takes in that state. Such a variable can be modeled with a function from the state-space \mathcal{S} to the set of real values \mathbb{R} . For instance, if we are interested in a variable called r we can model it as function $r : \mathcal{S} \rightarrow \mathbb{R}$. Then $r(X_n)$ indicates the value that this function takes in state X_n . We will call such functions *rewards* and say that reward $r(X_n)$ is earned when the system is in state X_n .

$$\text{A reward } r : \mathcal{S} \rightarrow \mathbb{R} \text{ is a function from } \mathcal{S} \text{ to } \mathbb{R} \quad (\text{B.3})$$

Notice that since X_n is a (discrete) random variable, $r(X_n)$ is a (discrete) random variable as well. Only when r is applied to one of the possible realizations of X_n (i.e. states in \mathcal{S}), a real value is obtained. The set of possible realizations of $r(X_n)$ is given by set $\mathcal{R} = \{r(i) \mid i \in \mathcal{S}\}$.

$$\begin{aligned} &\text{The probability that } r(X_n) \text{ takes value } v \in \mathcal{R}, \text{ written} \\ &P(r(X_n) = v), \text{ is given by } P(r(X_n) = v) = \sum_{i \in \mathcal{S}, r(i)=v} P(X_n = i) \end{aligned} \quad (\text{B.4})$$

Example B.2 (Game of dice continued). Consider the game of dice of Example B.1 again. Assume the player earns €1 when he or she throws a four, a five or a six, but has to pay €1 in case of a one, a two or a three. What is the probability that the player loses €1 in the fifth round and what is the probability that the player loses 3 consecutive rounds in a row? To answer these questions we model the amount earned or lost by a reward $r : \mathcal{S} \rightarrow \{-1, 1\}$ defined by $r(1) = r(2) = r(3) = -1$ and $r(4) = r(5) = r(6) = 1$. The probability that the player loses €1 in the fifth round is given by $P(r(X_4) = -1) = \sum_{i \in \mathcal{S}, r(i)=-1} P(X_4 = i) = P(X_4 = 1) + P(X_4 = 2) + P(X_4 = 3) = \frac{1}{2}$. The probability that the player loses 3 consecutive rounds equals the probability that $r(X_0) = -1$ and $r(X_1) = -1$ and $r(X_2) = -1$ and is written as $P(r(X_0) = -1, r(X_1) = -1, r(X_2) = -1)$. Since the outcomes of the different tosses are independent, random variables $r(X_0)$, $r(X_1)$ and $r(X_2)$ are independent as well implying that $P(r(X_0) = -1, r(X_1) = -1, r(X_2) = -1) = P(r(X_0) = -1)P(r(X_1) = -1)P(r(X_2) = -1) = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$.

$$-1)P(r(X_2) = -1) = \frac{1}{2}^3 = \frac{1}{8}.$$

As explained before, the reward obtain in state X_n is given by $r(X_n)$ which is a random variable. Often we are not interested in the random variable itself, but in its *expected value* denoted by $E(r(X_n))$. $E(r(X_n))$ is called the *expected reward* at time n . Intuitively, this expected reward represents the average of a large number of independent realizations of $r(X_n)$ and is defined as the probability-weighted average $\sum_{v \in \mathcal{R}} v \cdot P(r(X_n) = v)$. By using (B.4) it is easy to show that $E(r(X_n)) = \sum_{i \in S} r(i) \cdot P(X_n = i) = \sum_{i \in S} r(i) \cdot \pi_i^{(n)}$. Now assume we consider function r to be the row vector $[r(1), r(2), \dots, r(N)]$. Then $\sum_{i \in S} r(i) \cdot \pi_i^{(n)}$ can conveniently be written as $\pi^{(n)} r^T$, where $\pi^{(n)} r^T$ is the matrix product between row vector $\pi^{(n)}$ and column vector r^T . Thus

$$\text{The expected reward at time } n \text{ is given by } E(r(X_n)) = \pi^{(n)} r^T \quad (\text{B.5})$$

Example B.3 (Game of dice continued). Let us consider the game of dice again, continuing Examples B.1 and B.2. Assume we are interested in the expected reward obtained in the fifth round. This is given by $E(r(X_4)) = \pi^{(4)} r^T = [\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}] [-1, -1, -1, 1, 1, 1]^T = 0$. Apparently, but not unexpectedly, not a very lucrative game to play.

Exercise B.1 (Wealthy gambler - expected reward). A very wealthy gambler (having a practically unlimited amount of cash) is playing a game of roulette at the casino. The pockets of the roulette wheel are numbered from 0 to 36 where odd numbers are red, even numbers are black and number 0 is green. At each spin the gambler places €500 on red. If the outcome of the spin is red the player earns €500 (and receives the €500 on this game back) and otherwise the €500 are lost.

- (a) Model this game as stochastic process, explain how the state-space of the process is defined and give the probability distributions.
- (b) Define a reward function that models the amount of money earned or lost after each spin of the wheel.
- (c) Compute the expected reward earned after the fifth spin of the wheel.
- (d) Compute the probability that the player loses 20 times in a row.

See answer.

Exercise B.2 (Time-slotted Ethernetwork - throughput). Two senders, A and B , offer frames to a time-slotted Ethernetwork. During each time slot sender A offers a frame with probability $\frac{1}{3}$ and sender B offers a frame with probability $\frac{1}{5}$. When two frames are offered during the same time slot, both frames are lost. When precisely one

frame is offered, this frame is transmitted. When no frame is offered, the slot remains idle.

- (a) Model this communication system as a stochastic process, explain the state-space of the process and give the probability distribution.
- (b) Define a reward denoting the number of transmitted frame during a time slot.
- (c) Compute the probability that the medium does not transmit a frame during time slot n .
- (d) Compute the throughput of the medium, i.e. the expected number of transmitted frames per time slot.

See answer.

Exercise B.3 (Expected reward - computation). Show that equation (B.5) holds.

See answer.

B.2 Markov chains

B.2.1 Independent identically distributed variables

The game of dice in Examples B.1, B.2 and B.3 illustrates a sequence of X_0, X_1, X_2, \dots of *independent identically distributed* random variables. The variables are called *identically distributed* if they have the probability distributions, i.e. $\pi^{(n)} = \pi^{(m)}$ for all $n, m \geq 0$. They are called (pairwise²) *independent* if $P(X_n = i, X_m = j) = P(X_n = i)P(X_m = j)$ for all $n \neq m$ and $i, j \in \mathcal{S}$.

Another way of looking at the dependency between random variables is to consider *conditional probabilities*. The probability that X_n takes value i given the fact that X_m takes value j , written $P(X_n = i | X_m = j)$, is defined as

$$P(X_n = i | X_m = j) = \frac{P(X_n = i, X_m = j)}{P(X_m = j)} \quad (\text{when } P(X_m = j) > 0) \quad (\text{B.6})$$

Here $P(X_n = i, X_m = j)$ denotes the probability that both $X_n = i$ and $X_m = j$. Therefore

²Next to pairwise independence, the concept of mutual independence is defined in literature. The variables are called mutual independent if $P(X_{n_1} = i_1, X_{n_2} = i_2, \dots, X_{n_k} = i_k) = P(X_{n_1} = i_1) P(X_{n_2} = i_2) \dots P(X_{n_k} = i_k)$ for any finite sequence n_1, n_2, \dots, n_k (where $n_i < n_{i+1}$ ($i = 1, 2, \dots, k-1$)) and states $i_1, i_2, \dots, i_k \in \mathcal{S}$. Mutual independence implies pairwise independence, but not the other way around.

$$X_n \text{ and } X_m \text{ are independent if} \\ P(X_n = i | X_m = j) = P(X_n = i) \text{ for all } i, j \in \mathcal{S} \quad (\text{B.7})$$

Hence the probability that X_n takes value i does not depend on the value that X_m assumes.

B.2.2 Markovian property

Discrete-time stochastic processes with independent identically distributed variables are relatively easy to analyze. For instance, if the probability mass function of one variable is known the probability mass function of the other variables are known as well. Expected rewards can then readily be computed. In addition, the property of independence allows the application of powerful theorems, such as the strong law of large numbers and the central limit theorem (which we will discuss in later sections). Disadvantage is that the random behaviour of systems can often not be captured using independent identically distributed variables. For example, the number of packets present in a packet buffer during a certain time slot will typically depend on the number of packets present during the previous time slot. To model such behaviour we will study a class of discrete-time stochastic processes called *Markov chains*. Markov chains, named after the influential Russian mathematician Andrey Markov, are rich enough to capture dependencies between random variable but at the same time have sufficient structure to allow the analysis of interesting properties.

$$\begin{aligned} \text{A discrete-time stochastic processes } X_0, X_1, X_2, \dots \text{ is called a} \\ \text{Markov chain if for all } n \geq 0 \text{ and } i_0, \dots, i_{n-1}, i, j \in \mathcal{S} \\ P(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, \dots, X_0 = i_0) = \\ P(X_{n+1} = j | X_n = i) \end{aligned} \quad (\text{B.8})$$

This property is called the *Markovian property*. It states that the random state at time $n+1$ is determined completely by the state realized at time n , and is not depending on the states that were visited before time n . Usually (and we follow this convention in these notes) it is assumed that the Markov chain is *time-homogeneous* implying that for each pair $i, j \in \mathcal{S}$ a constant P_{ij} exists such that for all $n \geq 0$

$$P(X_{n+1} = j | X_n = i) = P_{ij} \quad (\text{B.9})$$

P_{ij} is the probability that state j is visited at time $n+1$ given that the system is in state i at time n . In other words P_{ij} is the probability that the system transits from state i to state j in a single step. Therefore P_{ij} is called the *one-step transition probability* from state i to state j . The property of time homogeneity makes this transition probability independent of the time the transition takes place.

Since $\mathcal{S} = \{1, 2, \dots, N\}$, the one-step transition probabilities can be ordered in a square $N \times N$ *transition probability matrix* P of the form

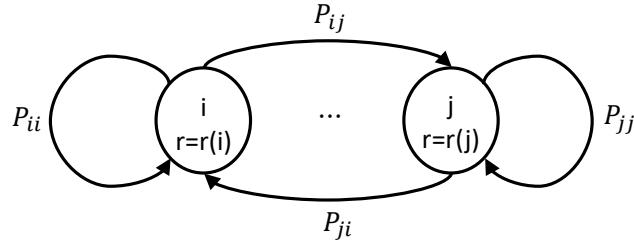


Figure B.2: Excerpt of a transition diagram

$$P = \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1N} \\ P_{21} & P_{22} & \cdots & P_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ P_{N1} & P_{N2} & \cdots & P_{NN} \end{bmatrix} \quad (\text{B.10})$$

So P_{ij} is the element positioned in *row* i and *column* j of the matrix P . It can be shown that the sum of the probabilities in each row of the transition probability matrix must equal one. Thus

$$\sum_{j \in \mathcal{S}} P_{ij} = 1 \text{ for all } i \in \mathcal{S} \quad (\text{B.11})$$

The information about the transition probability can nicely be represented graphically by a so-called *transition diagram*. This is a directed graph with N nodes, one node for each of the states of the Markov chain. Each node has a corresponding state label. Optionally, if a reward is defined, the value obtained in a state is denoted as well. Next to nodes the diagram has labeled directed arcs. A directed arc with label P_{ij} exists between node i and node j if $P_{ij} > 0$. Figure B.2 shows an excerpt of a transition diagram with two states, i and j . The arcs should only be drawn if the corresponding probabilities are greater than 0. In the figure, labels $r = r(i)$ and $r = r(j)$ imply that reward r is defined and take values $r(i)$ and $r(j)$ in states i and j respectively. A Markov chain for which such a reward is defined is sometimes called a *Markov reward model*.

Example B.4 (Gambler's ruin). Consider a gambler playing a game of roulette at the cassino, carrying €100 of cash. The pockets of the roulette wheel are numbered from 0 to 36 where odd numbers are red, even numbers are black and number 0 is green. At each spin the gambler places €100 on black. If the outcome of the spin is black the player wins €100 and if the outcome is red the player loses €100. When the gambler has either €0 or €300 in cash (s)he quits the game. When we assume the probability of hitting the green pocket to be zero, we can model this game as a Markov reward model with state space $\mathcal{S} = \{1, 2, 3, 4\}$. Each state corresponds to the amount of money the gambler has in cash. This amount can be modelled using a reward function $r : \mathcal{S} \rightarrow \{0, 100, 200, 300\}$, where $r(1) = 0, r(2) = 100, r(3) = 200$ and

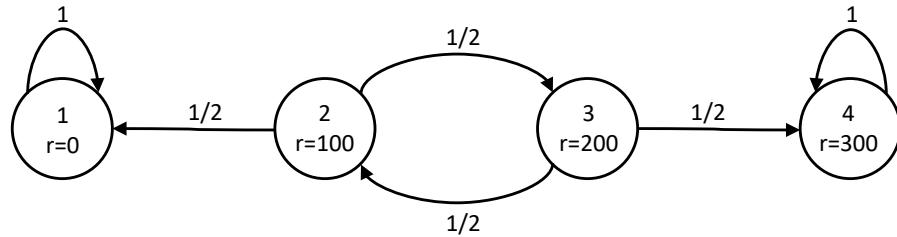


Figure B.3: Transition diagram of the gambler's ruin

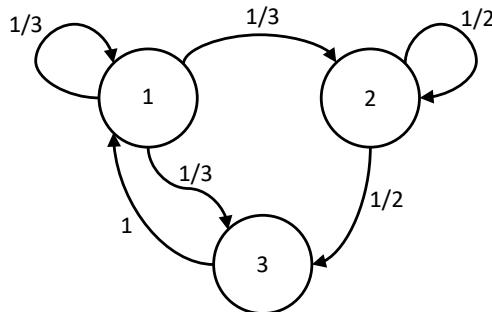


Figure B.4: Transition diagram of three-state Markov chain

$r(4) = 300$. Figure B.3 shows the transition diagram of this game. The corresponding probability matrix P is given by

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In the transition diagram the nodes (represented by circles) are labeled with the name of the state, as well as with the reward obtained in that state. Assume the player currently has €200 in cash. This corresponds to the node of the diagram labeled with (state) 3 and (reward) 200. Upon a spin of the wheel, a transition is made. The probability that the outcome of spin is black is $\frac{1}{2}$ and therefore an arc is drawn from state 3 to state 4 labeled with $\frac{1}{2}$. The Markov chain thus makes a transition from state 3 to state 4 when the outcome of the spin is black. After this transition, the player has €300 in cash. The fact that the player will then quite the game is modelled by transition probability $P_{44} = 1$.

Exercise B.4 (Transition diagram to matrix). Consider the transition diagram of the three-state Markov chain depicted in Figure B.4. Give the transition probability matrix of this chain.

See answer.

Exercise B.5 (Matrix to transition diagram). Draw the transition diagram corresponding the Markov chain with transition probability matrix

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{8} & \frac{2}{3} & \frac{1}{6} \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \end{bmatrix}$$

See answer.

Exercise B.6 (Markov chains - dependent and non-identically distributed variables). Consider a Markov chain X_0, X_1, \dots with two states, 1 and 2. At times $0, 2, \dots$ the process visits state 1 and at times $1, 3, \dots$ it visits state 2.

- (a) Give the transition probability matrix of this chain and draw the transition diagram.
- (b) Give the initial distribution at time 0, i.e. $\pi^{(0)}$.
- (c) Determine the distribution at time 1.
- (d) Show that X_0 and X_1 are not identically distributed.
- (e) Show that X_0 and X_1 are dependent variables.

See answer.

Exercise B.7 (Gambler's ruin - probability distributions). Consider Markov chain X_0, X_1, \dots corresponding to transition diagram of the gambler's ruin in Figure B.3 and assume $\pi^{(0)} = [0, 1, 0, 0]$.

- (a) Determine $\pi^{(1)}$.
- (b) Determine $\pi^{(2)}$.

See answer.

Exercise B.8 (Markov chains - independent identically distributed variables). Consider Markov chain X_0, X_1, \dots with state-space $\{1, 2\}$, initial distribution $\pi^{(0)} = [\frac{1}{2}, \frac{1}{2}]$ and transition probability matrix

$$P = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

- (a) Show that X_n and X_{n+1} are identically distributed for all $n = 0, 1, \dots$.
- (b) Show that X_n and X_{n+1} are independent for all $n = 0, 1, \dots$.

See answer.

B.3 Transient analysis

Now that we have defined Markov chains and Markov reward models, we will study how to analyze them. In particular we will study how to compute the probability distributions $\pi^{(n)}$ (B.2). When these are known, we can compute the expected reward (B.5) obtained at time n .

B.3.1 Transient analysis via matrix algebra

It can be shown that for all $i, j \in \mathcal{S}$ and $n, m \geq 0$, $P(X_n = j) = \sum_{i \in \mathcal{S}} P(X_n = j | X_m = i)P(X_m = i)$. By using (B.6) we then obtain the so-called *law of total probability* stating that for all $i, j \in \mathcal{S}$ and $n, m \geq 0$

$$P(X_n = j) = \sum_{i \in \mathcal{S}} P(X_n = j | X_m = i)P(X_m = i) \quad (\text{B.12})$$

In particular we have $P(X_{n+1} = j) = \sum_{i \in \mathcal{S}} P(X_{n+1} = j | X_n = i)P(X_n = i)$ and thus, by using (B.2) and (B.9), and we have for all $n \geq 0$,

$$\pi_j^{(n+1)} = \sum_{i \in \mathcal{S}} P_{ij}\pi_i^{(n)} \quad (\text{B.13})$$

So we obtain the j^{th} element of $\pi^{(n+1)}$ by taking the product of row vector $\pi^{(n)}$ with the j^{th} column of matrix P . In other words

$$\pi^{(n+1)} = \pi^{(n)}P \quad (\text{B.14})$$

By induction on n it is then easy to show that for all $n \geq 0$

$$\pi^{(n)} = \pi^{(0)}P^n \quad (\text{B.15})$$

Hence when the initial probability distribution π^0 is known, all other distributions can be computed by repeated matrix multiplication.

Example B.5 (Gambler's ruin continued). Consider Example B.4 of the gambler playing a game of roulette at the cassino again. Upon entrance of the cassino the player carries €100 and thus the Markov chain starts in state 2 with probability 1, i.e. $\pi^0 = [0, 1, 0, 0]$. Assume we would like to know the probability that the player has won the game after 10 spins of the wheel as well as the expected amount of cash the

player carries after 10 spins of the wheel. The probability is given by $P(r(X_{10}) = 300) = \{\text{according to (B.4)}\} P(X_{10} = 4) = \{\text{by (B.2)}\} \pi_4^{(10)} = \{\text{by (B.15)}\} (\pi^0 P^{10})_4$. Now

$$P^{10} \approx \begin{bmatrix} 1.000 & 0.000 & 0.000 & 0.000 \\ 0.667 & 0.001 & 0.000 & 0.333 \\ 0.333 & 0.000 & 0.001 & 0.667 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix}$$

and thus $\pi^0 P^{10} \approx [0.667, 0.001, 0.000, 0.333]$ and $(\pi^0 P^{10})_4 \approx 0.333$. The expected amount of cash after 10 spins is given by $E(r(X_{10})) = \{\text{by B.5}\} \pi^{(10)r^T} = \pi^0 P^{10}[0, 100, 200, 300]^T \approx [0.667, 0.001, 0.000, 0.333][0, 100, 200, 300]^T \approx €100$. The player can improve the odds of winning if (s)he begins the game with €200. In that case $\pi^0 = [0, 0, 1, 0]$, the probability of winning (after 10 spins) is about 0.667 and the expected amount of cash (after 10 spins) is about €200.

Based on the Markovian property (B.8) and the law of total probability (B.12) it can be shown that for each $m, n \geq 0, i, j \in \mathcal{S}$

$$P_{ij}^n = P(X_{m+n} = j \mid X_m = i) \quad (\text{B.16})$$

So P_{ij}^n is the probability that the system transits from state i to state j in n steps. Therefore P_{ij}^n is called the *n-step transition probability* from state i to state j and P^n is called the *n-step transition probability matrix*. Notice that equation (B.16) for the n-step probabilities generalizes equation (B.9) for the one-step transition probabilities. Notice further that $P^0 = I$, where I denotes the $N \times N$ identity matrix, implying that $P_{ii}^0 = 1$ and $P_{ij}^0 = 0$ when $i \neq j$.

Exercise B.9 (Probability distributions via matrix algebra). Consider Markov chain X_0, X_1, \dots with initial distribution $\pi^{(0)} = [1, 0]$ and transition probability matrix

$$P = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ 1 & 0 \end{bmatrix}$$

- (a) Compute $P(X_1 = 1)$.
- (b) Compute $P(X_2 = 1)$.
- (c) Approximate $P(X_{20} = 1)$.
- (d) Approximate the probability that the Markov chain is in state 1 after a very large number of transitions.

See answer.

B.3.2 Transient analysis via paths

To give more insight in the meaning of P_{ij}^n we will introduce the notion of path.

A sequence i_1, i_2, \dots, i_n of states (where $n \geq 1$)
is called a *path* if $P_{i_m i_{m+1}} > 0$ for each $m \in \{1, 2, \dots, n-1\}$

(B.17)

A path i_1, i_2, \dots, i_n is said to be a path from i_1 to i_n having length $n - 1$, where the length refers to the number of transitions made. The probability of a path i_1, i_2, \dots, i_n , written $P(i_1, i_2, \dots, i_n)$, is defined as

$$P(i_1, i_2, \dots, i_n) = \prod_{m=1}^{n-1} P_{i_m i_{m+1}}$$
(B.18)

Here we follow the usual convention that $\prod_{m=1}^{n-1} P_{i_m i_{m+1}} = 1$ if $n = 1$.

We can now express the n-step transition probability P_{ij}^n in terms of the paths from i to j of length n , which is done in the following equation

$$P_{ij}^n = \sum \{P(i, i_1, \dots, i_{n-1}, j) \mid i, i_1, \dots, i_{n-1}, j \text{ is a path of length } n\}^a$$
(B.19)

^aIn case different paths have the same probability, these probabilities have to be individually accounted for. Therefore this set is a multiset.

So P_{ij}^n can be computed by summing up the probabilities of each path from i to j of length n . Notice that for $n = 0$ and $i = j$, exactly one path exists (namely single-state path i) having probability 1. If $n = 0$ and $i \neq j$, no path exists of length 0 resulting in a sum over an empty set which equals 0. For $n = 1$ we obtain path i, j having probability P_{ij} .

From matrix algebra we know that for all $n, m \geq 0$, $P^{m+n} = P^m P^n$. This implies that for each $i, j \in \mathcal{S}$, P_{ij}^{m+n} is obtained by multiplying the i^{th} row of P^m with the j^{th} column of P^n , i.e. that

$$P_{ij}^{m+n} = \sum_{k \in S} P_{ik}^m P_{kj}^n$$
(B.20)

The equations in (B.20) are called the *Chapman-Kolmogorov equations*³. Thus the probability that an $(m + n)$ -step transition from state i to state j is made is the sum over all states k of the probability that an m -step transition is made from i to k times the probability that an n -step transition is made from k to j .

³This is in fact the case if P_{ij}^n are indeed interpreted as conditional probabilities as given in (B.16).

Example B.6 (Gambler's ruin continued). Consider matrix P^{10} of Example B.5 again. The probability that the gambler will win the game after precisely 10 spins, starting with an initial capital of €100, is given by $P_{2,4}^{10}$ which is approximately 0.333. The gambler can double the odds of winning by taking €200 as initial capital, as can be learned by inspecting $P_{3,4}^{10}$. $P_{3,4}^{10}$ can be computed by repeated matrix multiplication, but also by considering the paths of length 10 from state 3 to state 4. There exist five such paths:

- path 3, 4, 4, 4, 4, 4, 4, 4, 4, 4 with probability $\frac{1}{2}$
- path 3, 2, 3, 4, 4, 4, 4, 4, 4, 4 with probability $\frac{1}{2}^3$
- path 3, 2, 3, 2, 3, 4, 4, 4, 4, 4 with probability $\frac{1}{2}^5$
- path 3, 2, 3, 2, 3, 2, 3, 4, 4, 4 with probability $\frac{1}{2}^7$
- path 3, 2, 3, 2, 3, 2, 3, 4, 4 with probability $\frac{1}{2}^9$

As expected, the sum of their probabilities equals $\frac{1}{2} + \frac{1}{2}^3 + \frac{1}{2}^5 + \frac{1}{2}^7 + \frac{1}{2}^9 \approx 0.667$. When starting with €200 the probability that the gambler has €100 in his/her pocket after 10 spins is *precisely* 0. The reason is that Markov chain, when starting in state 3 can never be in state 2 after precisely 10 transitions. A path of length 10 from state 3 to state 2 does not exist.

Exercise B.10 (N-step transition probabilities). Consider a Markov chain with transition probability matrix

$$P = \begin{bmatrix} \frac{1}{5} & \frac{4}{5} \\ 1 & 0 \end{bmatrix}$$

- Approximate the probability that the chain transits from state 1 to state 2 in precisely 5 steps using the technique of matrix multiplication (see also equation (B.16)).
- Approximate the probability that the chain transits from state 1 to state 2 in precisely 5 steps, by summing of the probabilities of paths (see also equation (B.19)).

See answer.

Exercise B.11 (Gambler's ruin - n-step probabilities and expected reward). Consider Examples B.4, B.5 and B.6 of the gambler's ruin.

- Approximate the probability that the gambler is broke after 100 spins, starting with an initial capital of €200.

- (b) Compute the probability that the gambler has €200 cash after 1000 spins, starting with an initial capital of €100.
- (c) Assume $\pi^{(0)} = [0, \frac{1}{2}, \frac{1}{2}, 0]$. Approximate the expected amount of cash the gambler is carrying after 16 spins of the wheel.

See answer.

Exercise B.12 (Gambler's ruin - dependent and non-identically distributed variables). Consider the Markov chain X_0, X_1, \dots of the gambler's ruin in Example B.4, where $\pi^{(0)} = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$.

- (a) Show that X_3 and X_6 are not identically distributed.
- (b) Show that X_3 and X_6 are dependent variables.

See answer.

Exercise B.13 (Queue in time-slotted communication network - modeling and transient analysis). Consider a queue in a time-slotted communication network. The queue can store up to 3 packets. During a time slot a packet is offered to the queue with probability $\frac{1}{3}$ in case the buffer is not full and retrieved from the queue with probability $\frac{1}{4}$ in case the buffer is not empty. Initially, the queue is empty.

- (a) Model the behaviour of the queue as a Markov reward model and specify the probability matrix, the initial distribution, and the reward function. Consider as states the queue occupancy at the beginning of a time slot.
- (b) Approximate the probability that the queue is full at the beginning of time slot 15.
- (c) Assume that the queue contains 2 packets at the beginning of time slot 20. Approximate the probability that the queue is empty at the beginning of time slot 40.
- (d) Approximate the expected queue occupancy at the beginning of time slot 5.

See answer.

Exercise B.14 (Independent identically distributed variables as Markov chain). Let X_0, X_1, \dots be a discrete-time stochastic process with state space $\{1, 2\}$ and independent identically distributed variables.

- (a) Show that this process is in fact a Markov chain by constructing a transition probability matrix that captures the intended behaviour.

- (b) Show that the random variables of the Markov chain X_0, X_1, \dots defined in (a) are indeed independent and identically distributed.

See answer.

B.4 State classification

In the previous section we studied how to compute probability vectors $\pi^{(n)}$ and expected rewards $E(r(X_n))$. These involve *transient properties* concerning the behaviour the Markov reward model has displayed after a fixed number of steps. Analyzing such properties is sometimes called *transient analysis*. In the next sections we will perform *long-run analysis* considering the *limiting behaviour* displayed after a very large number of steps. The analysis of the limiting behaviour depends on the state structure of the Markov chain, which is the topic of this section.

B.4.1 Communicating states

We will first define an equivalence relation over the states of a Markov chain.

$$\begin{aligned} \text{A state } j \text{ is } &\text{accessible from } i, \text{ written } i \rightarrow j, \\ &\text{if and only if } P_{ij}^n > 0 \text{ for some } n \geq 0 \end{aligned} \tag{B.21}$$

Note that $P_{ii}^0 = 1$ and thus $i \rightarrow i$ for every $i \in \mathcal{S}$. Further note that $i \rightarrow j$ if and only if there exists a path from i to j .

$$\begin{aligned} \text{States } i \text{ and } j \text{ communicate, written } i \leftrightarrow j, \\ &\text{if and only if } i \rightarrow j \text{ and } j \rightarrow i \end{aligned} \tag{B.22}$$

It is not hard to show that \leftrightarrow is an *equivalence relation*, a relation that is reflexive, symmetric and transitive. Hence for all $i, j, k \in \mathcal{S}$

- $i \leftrightarrow i$ (reflexive);
- if $i \leftrightarrow j$ then $j \leftrightarrow i$ (symmetric);
- if $i \leftrightarrow j$ and $j \leftrightarrow k$ then $i \leftrightarrow k$ (transitive).

$$\text{Relation } \leftrightarrow \text{ is an equivalence relation} \tag{B.23}$$

Since \leftrightarrow is an equivalence relation it *partitions* the state space \mathcal{S} into so-called *classes* of states. These classes are pair-wise disjoint implying that the intersection of two different classes is empty. In addition each class is non-empty and the union of all classes together equals the set \mathcal{S} of all states.

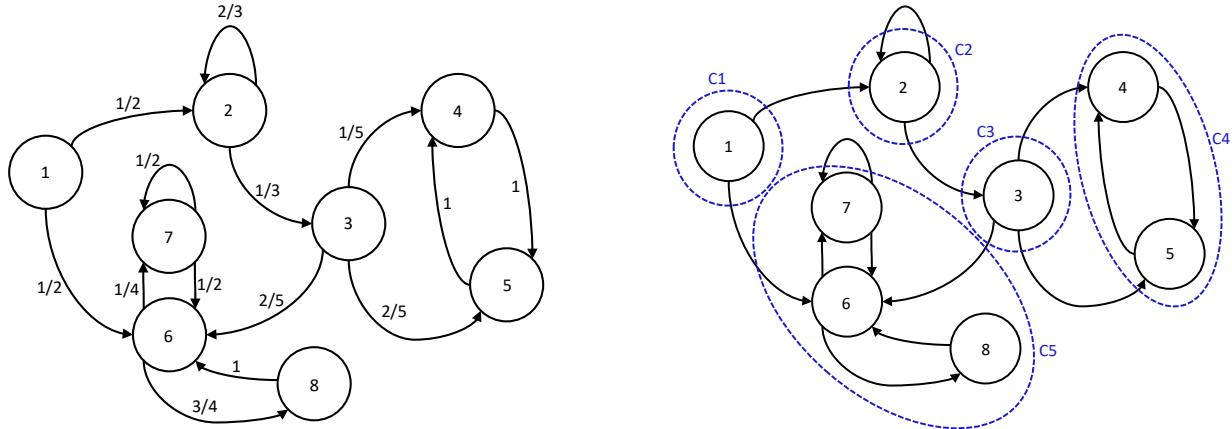


Figure B.5: Classes of communicating states depicted in transition diagram

Example B.7 (Classes of communicating states). Consider the transition diagram of the Markov chain depicted at the left of Figure B.5. The classes of communicating states are shown at the right of the Figure by means of dashed ellipses in blue color and have (arbitrary) names C_1, C_2, C_3, C_4 and C_5 (also indicated in blue color). To avoid clutter, the probabilities on the arcs are emitted from the diagram. Notice that every state is accessible from state 1. On the other hand state 1 cannot be accessed from any state other than itself. Therefore state 1 only communicates with itself and is contained in a singleton class $C_1 = \{1\}$. The same holds for states 2 and 3 which are contained in classes $C_2 = \{2\}$ and $C_3 = \{3\}$ respectively. State 5 is accessible from state 4 in a single step and, vice versa, state 4 is accessible from state 5 in a single step. Therefore states 4 and 5 communicate and are element of the same class C_4 . No state other than 4 is accessible from 5 and therefore C_4 contains no other states. Finally, states 6 and 7 communicate and so do 6 and 8. Because of transitivity states 7 and 8 therefore communicate as well (which is also immediately clear from the diagram). Together states 6, 7 and 8 form class C_5 .

Exercise B.15 (Classes of a Markov chain). Consider a Markov chain with state

space $\{1, 2, 3, 4, 5\}$ and transition probability matrix $P = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$. Determine the classes of this Markov chain.

See answer.

Exercise B.16 (State accessibility versus paths). Show that $i \rightarrow j$ if and only if there exists a path from i to j .

See answer.

Exercise B.17 (Communicating states - equivalence relation). Show that relation \leftrightarrow is an equivalence relation (see equation (B.23)).

See answer.

B.4.2 Recurrent and transient states

As a next concept we introduce the notion of recurrency.

A state i is said to be *recurrent* if and only if $f_{ii} = 1$ (B.24)

Here f_{ii} denotes the *return probability* that the chain, starting from i , eventually revisits i in one or more steps. This return probability f_{ii} is formally defined in terms of the paths from i to i with a length of at least 1:

$$f_{ii} = \sum \{ P(i, i_1, \dots, i_{n-1}, i) \mid i, i_1, \dots, i_{n-1}, i \text{ is a path of length } n \geq 1 \text{ such that } i_k \neq i \text{ for all } k = 1, \dots, n-1 \} \quad (\text{B.25})$$

Although this definition can be used to compute return probabilities, this can quickly become very complicated. In the next section we will show how to effectively compute these probabilities by solving a system of linear equations.

When a chain ever enters a recurrent state, it will eventually return to that state with probability 1 and will thus keep returning to it infinitely often. A state that is not recurrent is called *transient*. Such a transient state is only visited a finite number of times, implying that at some point in time the chain will not visit such a state anymore.

A state i is said to be *transient* if and only if $f_{ii} < 1$ (B.26)

It can be shown that the states in any class of a Markov chain are either all recurrent or all transient. Hence we can and will safely talk about classes being recurrent or transient.

The properties of recurrence and transience are class properties (B.27)

It can be shown that a class is recurrent if and only if none of its states has a transition leading to a state outside the class⁴. Such a class is called *closed*. A class that is not closed is transient, implying that at least one of its states has a transition leading (with a positive probability) to a state outside the class. More formally, let if $C \subseteq \mathcal{S}$ denotes a class of states, then

C is recurrent if and only if for all $i \in C$ and $j \in \mathcal{S} \setminus C$, $P_{ij} = 0$ (B.28)

⁴This is true for Markov chains with a finite state space as we consider in these notes. For infinite state systems, this property does not hold in general.

For reasons of brevity we will occasionally explain the behaviour of a chain at the class level instead of at the state level. We will for instance say that a chain can transition from one class to another if a transition between corresponding states exists, or we will say that the Markov chain visits a class to mean that the chain visits one of the states in that class.

Example B.8 (Recurrent versus transient classes). Consider again the transition diagrams in Figure B.5. A transition exists from class C_1 to class C_2 and therefore class C_1 is transient. It is evident from the figure that the Markov chain can never visit class C_1 more than once. When this class was visited the chain either transitions to C_2 or C_5 . Class C_2 is also transient since it has a transition to class C_3 . Although the chain can repeatedly visit C_2 , at some point in time it will escape and move forward to class C_3 . Also this class is transient since a transition can be made to either class C_4 or C_5 . C_4 and C_5 do not have outgoing transitions and therefore these classes are both recurrent. No matter the starting state of the chain, eventually (with probability one) it will end up in either C_4 or C_5 and remain there for ever.

Exercise B.18 (Recurrent versus transient classes). Consider a Markov chain

$$\text{with state space } \{1, 2, 3, 4, 5\} \text{ and transition probability matrix } P = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Determine for each of the classes and states whether they are recurrent or transient.

See answer.

Exercise B.19 (Computing return probabilities through paths). Consider a Markov chain with state space $\{1, 2, 3, 4, 5\}$ and transition probability matrix $P =$

$$\begin{bmatrix} \frac{1}{3} & \frac{2}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

- (a) State 1 is transient. Show this by computing the return probability f_{11} .
- (b) State 2 is recurrent. Show this by computing the return probability f_{22} .

See answer.

B.4.3 Periodic and aperiodic states

As a final concept in this section we will introduce the notion of periodicity for recurrent states⁵.

The *period* of recurrent state i , written $d(i)$, is defined as

$$d(i) = \gcd\{n \geq 1 \mid P_{ii}^{(n)} > 0\} \quad (\text{B.29})$$

Here $\gcd\{n \geq 1 \mid P_{ii}^{(n)} > 0\}$ denotes the *greatest common divisor* of set $\{n \geq 1 \mid P_{ii}^{(n)} > 0\}$. Since $P_{ii}^{(n)} > 0$ if and only if a path of length n exists between i and i we can characterise $d(i)$ in an alternative way as

$$d(i) = \gcd\{n \geq 1 \mid \text{a path of length } n \text{ exists from } i \text{ to } i\} \quad (\text{B.30})$$

Notice that since i is recurrent $f_{ii} > 0$ and thus (by equation (B.25)) at least one path with positive length exists between i and i . Hence the sets specified in (B.29) and (B.30) are non-empty and the \gcd is well-defined. To compute the period of a state i using (B.30), we only have to consider paths that do not visit state i in between.

We thus have that for each recurrent state i the number of steps for which return to i is possible, is always a multiple of the period $d(i)$. Stated otherwise, returning in m steps is *not possible* in case m is not a multiple of $d(i)$. Hence in case state i is entered at time n this state cannot be visited at times $n + m$ if m is not a multiple of $d(i)$. In case $d(i) \geq 2$ this implies that certain revisiting times are excluded (e.g. $n + 1$). States for which this holds are called *periodic* states. Such states are always visited in periodic patterns, dependent on the time of entry of the state. One could say that the Markov chain in some way memorizes the entry time of a periodic state. In case $d(i) = 1$ no revisiting times are excluded. States for which this holds are called *aperiodic* states. One can show that, when an aperiodic state is entered at time n , from time $n + (N - 1)^2 + 1$ onwards this state has a positive probability of being revisited. The memory about the entry time appears to fade away.

A recurrent state i is called *aperiodic* if $d(i) = 1$
and *periodic* if $d(i) > 1$. (B.31)

It can be shown that all states in the same recurrent class have the same period. In particular the states in a recurrent class are either all periodic or all aperiodic. We therefore often discuss periodicity at the class level and speak about periodic or aperiodic classes.

Periodicity is a class property;
all states in a recurrent class have the same period (B.32)

⁵One can also define this notion for transient states, but we will not require that for our purposes. In the sequel we will assume a state to be recurrent when we refer to its period.

Earlier we claimed that an aperiodic state has a positive probability of being revisited at any time after a certain fixed time period has passed. This property is a special case of the following fact:

$$\begin{aligned} \text{For any aperiodic class } C \text{ and all states } i, j \in C, \\ P_{ij}^n > 0 \text{ for all } n \geq (N - 1)^2 + 1 \end{aligned} \quad (\text{B.33})$$

Hence when a Markov chain enters a recurrent aperiodic class at time n , from moment $n + (N - 1)^2 + 1$ onwards, each state in this class has a positive probability of being visited. The memory about the entrance state and time disappear.

Example B.9 (Periodic versus aperiodic classes). Consider again the transition diagrams in Figure B.5. Classes C_4 and C_5 are recurrent. Let us consider these classes in turn, starting with class C_4 . To understand whether C_4 is periodic or aperiodic we can pick any state, say state 4. Now it is easy to see that $P_{44}^n = 0$ if n is odd and $P_{44}^n = 1$ if n is even. In other words, the lengths of the paths from state 4 to state 4 are $2, 4, 6, \dots$ with greatest common divisor 2. Therefore $d(4) = 2$ and class C_4 is periodic. When the Markov chain enters state 4 at time 0, it will revisit this state at times $2, 4, 6, \dots$ and visit state 5 at times $1, 3, 5, \dots$. Notice that since $d(4) = 2$, $d(5) = 2$ as well. For class C_5 consider state 7. Since $P_{77}^1 = \frac{1}{2}$, $d(7)$ must be 1. But then also $d(6) = 1$ and $d(8) = 1$ and so C_5 is an aperiodic class. Notice that if any state in a recurrent class has a transition to itself, the corresponding class must be aperiodic.

We conclude this section with some terminology we will use in these notes⁶. A class that is both recurrent and aperiodic is called an *ergodic class*. A *Markov chain* is called *ergodic* if all its recurrent classes are *ergodic*. A *Markov chain* that is not ergodic is called *non-ergodic*. Such a chain thus has at least one periodic recurrent class.

An *unichain* is a Markov chain that contains a single recurrent class in addition to zero or more transient classes. A unichain visits its transients states a finite number of times, after which the chain enters the unique class of recurrent states in which it remains for ever. We will call a Markov chain that is not a unichain a *non-unichain*. Such a non-unichain thus has zero or more transient classes and two or more recurrent classes. The states in the transient classes will be visited a finite number of times, and eventually the chain will end up in one of the recurrent classes in which it will be trapped forever.

In Section B.6 we will study the long-run (or limiting) behaviour of Markov chains. It will appear that this long-run behaviour depends on *type* of the Markov, i.e. whether it is ergodic or non-ergodic and whether it is a unichain or a non-unichain. These types are summarized in Table B.1.

⁶Notice that other literature can use the terminology slightly differently. E.g. the term *ergodic* is occasionally reserved for irreducible Markov chains (i.e. Markov chains with a single recurrent class) with only aperiodic states.

| Markov chain type | unichain | non-unichain |
|-------------------|--|--|
| ergodic | a single recurrent class which is aperiodic (and zero or more transient classes) | at least two recurrent classes, all of which are aperiodic (and zero or more transient classes) |
| non-ergodic | a single recurrent class which is periodic (and zero or more transient classes) | at least two recurrent classes, which are not all aperiodic (and zero or more transient classes) |

Table B.1: Markov chain types

Example B.10 (Gambler's ruin continued). Consider the example of the gambler's ruin again, depicted in Figure B.3. This Markov chain has three classes of communicating states: $\{1\}$, $\{2, 3\}$ and $\{4\}$. Class $\{2, 3\}$ is a transient class. Classes $\{1\}$ and $\{4\}$ are both ergodic (i.e. recurrent and aperiodic). The chain is thus an ergodic non-unichain.

Exercise B.20 (Periodic versus aperiodic classes). Consider a Markov chain with

$$\text{state space } \{1, 2, 3, 4, 5\} \text{ and transition probability matrix } P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

- (a) Determine for each recurrent class whether it is periodic or aperiodic.
- (b) Is this Markov chain a unichain?

See answer.

Exercise B.21 (Aperiodic states are eventually visited). Consider a Markov

$$\text{chain with state space } \{1, 2, 3\} \text{ and transition probability matrix } P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}.$$

- (a) Determine the classes of this chain, determine for each class whether it is tran-

sient or recurrent and periodic or aperiodic.

- (b) Argue that P^4, P^5, P^6, \dots contain only positive elements.

See answer.

B.5 Hitting probabilities

In the previous section we defined the return probability f_{ii} . This probability is a special case of what is called the *hit probability* f_{ij} , i.e. the probability that the chain, starting from state i will ever (in one or more steps) reach state j . In this section we will define the probabilities to hit a state or a set of states, and show how to effectively compute them. In addition we will focus on the expected reward earned, until a state or set of states is hit.

B.5.1 Probability to hit a state

We define f_{ij} as the probability that state j is ever visited, starting from state i . f_{ij} is sum of the probabilities of all paths from i to j with a length of at least one, and not visiting state j in between:

$$f_{ij} = \sum \{P(i, i_1, \dots, i_{n-1}, j) \mid i, i_1, \dots, i_{n-1}, j \text{ is a path of length } n \geq 1 \text{ such that } i_k \neq j \text{ for all } k = 1, \dots, n-1\} \quad (\text{B.34})$$

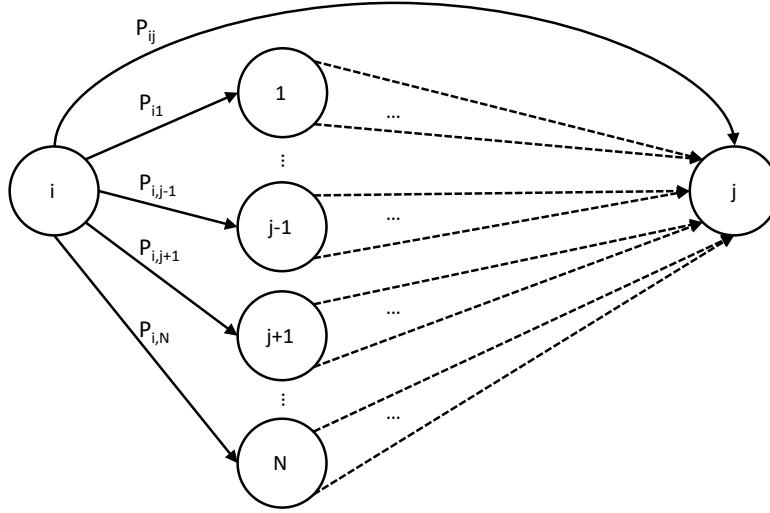
In general (B.34) does not yield a practical way of computing f_{ij} . Therefore we will establish a more convenient way.

Let us fix a state $j \in \mathcal{S}$ and define for each state $i \in \mathcal{S}$ a corresponding variable x_i (representing the hit probability f_{ij}). Consider the system of linear equations defined by

$$x_i = P_{ij} + \sum_{k \in \mathcal{S} \setminus \{j\}} P_{ik} x_k \quad (\text{B.35})$$

We claim that these equations are solved by substituting the values f_{ij} for the variables x_i . To see this, consider the set of paths leading from i to j (consistent with (B.34)) as depicted in Figure B.6. As shown in the figure, this set (having total probability f_{ij}) can be decomposed in the following subsets:

- a set with a single path i, j with total probability P_{ij}
- a set of paths of the form $i, 1, \dots, j$ with total probability $P_{i1} \cdot f_{1j}$
- ...
- a set of paths of the form $i, j-1, \dots, j$ with total probability $P_{i,j-1} \cdot f_{j-1,j}$

Figure B.6: Decomposition of set of paths from i to j

- ...
- a set of paths of the form $i, j+1, \dots, j$ with total probability $P_{i,j+1} \cdot f_{j+1,j}$
- ...
- a set of paths of the form i, N, \dots, j with total probability $P_{iN} \cdot f_{Nj}$

Hence $f_{ij} = P_{ij} + P_{i1} \cdot f_{1j} + \dots + P_{i,j-1} \cdot f_{j-1,j} + \dots + P_{i,j+1} \cdot f_{i,j+1} + \dots + P_{iN} \cdot f_{Nj} = P_{ij} + \sum_{k \in S \setminus \{j\}} P_{ik} f_{kj}$. Hence the equations in (B.35) are satisfied when variables x_i are replaced by values f_{ij} .

The system of linear equations (B.35) does not necessary have a unique solution, but the hitting probabilities form the least non-negative solution:

The hitting probabilities f_{ij} form the least non-negative solution to the following set of equations: $x_i = P_{ij} + \sum_{k \in S \setminus \{j\}} P_{ik} x_k$ (B.36)

We will show why (B.36) holds. To this end assume that $y_i \geq 0$ is a solution to the equations. Then

$$\begin{aligned} y_i &= P_{ij} + \sum_{k_1 \in S \setminus \{j\}} P_{ik_1} y_{k_1} \\ &= P_{ij} + \sum_{k_1 \in S \setminus \{j\}} P_{ik_1} (P_{k_1 j} + \sum_{k_2 \in S \setminus \{j\}} P_{k_1 k_2} y_{k_2}) \\ &= P_{ij} + \sum_{k_1 \in S \setminus \{j\}} P_{ik_1} P_{k_1 j} + \sum_{k_1 \in S \setminus \{j\}} \sum_{k_2 \in S \setminus \{j\}} P_{ik_1} P_{k_1 k_2} y_{k_2} \end{aligned}$$

Repeated substitution yields

$$\begin{aligned} y_i &= P_{ij} + \sum_{k_1 \in S \setminus \{j\}} P_{ik_1} P_{k_1 j} + \sum_{k_1 \in S \setminus \{j\}} \sum_{k_2 \in S \setminus \{j\}} \sum_{k_3 \in S \setminus \{j\}} P_{ik_1} P_{k_1 k_2} P_{k_2 j} + \dots + \\ &\quad \sum_{k_1 \in S \setminus \{j\}} \sum_{k_2 \in S \setminus \{j\}} \dots \sum_{k_n \in S \setminus \{j\}} P_{ik_1} P_{k_1 k_2} \dots P_{k_n j} + \\ &\quad \sum_{k_1 \in S \setminus \{j\}} \sum_{k_2 \in S \setminus \{j\}} \dots \sum_{k_n \in S \setminus \{j\}} \sum_{k_{n+1} \in S \setminus \{j\}} P_{ik_1} P_{k_1 k_2} \dots P_{k_n k_{n+1}} y_{k_{n+1}} \end{aligned}$$

Since $y_{k_{n+1}} \geq 0$, the last term is at least zero and thus

$$y_i \geq P_{ij} + \sum_{k_1 \in \mathcal{S} \setminus \{j\}} P_{ik_1} P_{k_1 j} + \sum_{k_1 \in \mathcal{S} \setminus \{j\}} \sum_{k_2 \in \mathcal{S} \setminus \{j\}} P_{ik_1} P_{k_1 k_2} P_{k_2 j} + \cdots + \sum_{k_1 \in \mathcal{S} \setminus \{j\}} \sum_{k_2 \in \mathcal{S} \setminus \{j\}} \cdots \sum_{k_n \in \mathcal{S} \setminus \{j\}} P_{ik_1} P_{k_1 k_2} \cdots P_{k_n j}$$

Now the first term in this inequality is the probability that the chain visits state j in precisely 1 transition. The second term is the probability that j is visited in exactly 2 transitions, etcetera, and the last term is the probability that j is visited in exactly $n+1$ transitions. For $n \rightarrow \infty$ the right-hand side of the inequality thus converges to f_{ij} and therefore $y_i \geq f_{ij}$.

The system of equations in (B.35) can have multiple solutions. This happens in case variables x_i are used that correspond to states i from which j cannot be reached. From (B.34) it follows that $f_{ij} = 0$ in these cases. Hence such variables x_i can be safely set to 0, consistent with finding the least non-negative solution as stated in (B.36).

Example B.11 (Hitting probabilities - hitting a state). Consider again the transition diagrams in Figure B.5. Assume we like to compute the probabilities to hit recurrent state 6. We can start considering only the states in the same recurrent class as 6 and obtain the equations $x_6 = \frac{3}{4}x_8 + \frac{1}{4}x_7$, $x_7 = \frac{1}{2} + \frac{1}{2}x_7$ and $x_8 = 1$. Solving these equation yields $x_6 = x_7 = x_8 = 1$, and thus $f_{66} = f_{76} = f_{86} = 1$. The fact that $f_{66} = 1$ is expected since 6 is a recurrent state. Since states 4 and 5 are in another recurrent class we have $x_4 = x_5 = 0$ and thus $f_{46} = f_{56} = 0$. Next we can compute x_3 as $x_3 = \frac{2}{5}$ and x_2 as $x_2 = \frac{2}{3}x_2 + \frac{1}{3}x_3 = \frac{2}{3}x_2 + \frac{2}{15}$, so $x_2 = \frac{2}{5}$. Hence $f_{26} = \frac{2}{5}$ and $f_{36} = \frac{2}{5}$. Finally $x_1 = \frac{1}{2}x_2 + \frac{1}{2} = \frac{1}{2} \cdot \frac{2}{5} + \frac{1}{2} = \frac{7}{10}$. Hence the probability that the Markov chain will ever hit state 6 starting from state 1, i.e. f_{16} , equals $\frac{7}{10}$. As another example consider the probabilities to hit transient state 3. Now state 3 can only be accessed from states 1 and 2 and therefore we only have to consider those states. We get $x_2 = \frac{1}{3} + \frac{2}{3}x_2$ and $x_0 = \frac{1}{2}x_2$. Solving yields $x_2 = 1$ and $x_1 = \frac{1}{2}$ and thus $f_{13} = \frac{1}{2}$ and $f_{23} = 1$. Notice that $f_{33} = 0$, which is consistent with the fact that state 3 is transient. It is illustrative to also compute f_{13} in terms of paths. An infinite number of paths exist from state 1 to state 3 (and not visiting state 3 in between):

- path 1, 2, 3 with probability $\frac{1}{2} \cdot \frac{1}{3}$
- path 1, 2, 2, 3 with probability $\frac{1}{2} \cdot \frac{2}{3}^1 \cdot \frac{1}{3}$
- path 1, 2, 2, 2, 3 with probability $\frac{1}{2} \cdot \frac{2}{3}^2 \cdot \frac{1}{3}$
- path 1, 2, 2, 2, 2, 3 with probability $\frac{1}{2} \cdot \frac{2}{3}^3 \cdot \frac{1}{3}$
- ...

Therefore, consistent with the computation above, $f_{13} = \sum_{n=0}^{\infty} \frac{1}{2} \cdot \frac{2}{3}^n \cdot \frac{1}{3} = \frac{1}{6} \sum_{n=0}^{\infty} \frac{2}{3}^n =^a \frac{1}{6} \frac{1}{1-\frac{2}{3}} = \frac{1}{2}$.

^aFor $|x| < 1$, $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$.

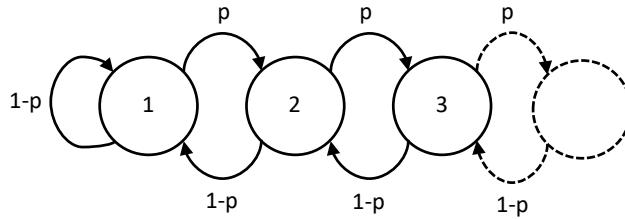


Figure B.7: Transition diagram of infinite Markov chain

Exercise B.22 (Computing return probabilities through equations). Consider a Markov chain with state space $\{1, 2, 3, 4, 5\}$ and transition probability matrix $P =$

$$\begin{bmatrix} \frac{1}{3} & \frac{2}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

- (a) Compute f_{11} by solving the system of linear equations in (B.36). Compare the result to the answer of Exercise B.19(a).
- (b) Compute f_{22} by solving the system of linear equations in (B.36). Compare the result to the answer of Exercise B.19(b).

See answer.

Exercise B.23 (Infinite closed classes are not necessarily recurrent). Equation (B.28) states that every closed class is recurrent. This is true for finite-state systems, but not necessarily for infinite-state systems, as will see in this exercise. To this end consider the transition diagram in Figure B.7 and assume $0 < p < 1$. Since all states communicate, we are dealing with a single class of states which is closed.

- (a) Assume $p = \frac{1}{2}$. Show that the class is recurrent by proving that $f_{11} = 1$. Hint: use Equation B.36 and exploit the fact that the hitting probability forms the least non-negative solution.
- (b) Assume $p = \frac{2}{3}$. Show that the class is transient by proving that $f_{11} < 1$. Hint: use Equation B.36 and show that $x_1 = \frac{2}{3}$ and $x_n = \frac{1}{2^n}$ (for $n \geq 2$) is a solution.

See answer.

B.5.2 Expected cumulative reward until hitting a state

Now that we can compute the probability to hit a state, it is interesting to also know the expected reward that is earned until this state is hit. We will define the expected cumulative

reward earned until state j is hit, starting from state i , as the probability-weighted average of the cumulative rewards of the paths (of length one or more) from i to j (not visiting j in between and not counting the reward obtained in the target state j), normalized to the probability that state j is actually hit.

The expected cumulative reward earned until state j
is hit, starting from state i , is defined as $\frac{f_{ij}^r}{f_{ij}}$

Recall from (B.34) that f_{ij} equals the probability that the chain, starting from state i will ever (in one or more steps) reach state j . f_{ij}^r is the probability-weighted average and is defined in terms of the paths from i to j :

$$f_{ij}^r = \sum_{\substack{i, i_1, \dots, i_{n-1}, j \text{ is a path of length } n \geq 1 \text{ such that} \\ i_k \neq j \text{ for all } k = 1, \dots, n-1}} \{P(i, i_1, \dots, i_{n-1}, j) \cdot (r(i) + r(i_1) + \dots + r(i_{n-1}))\}$$

This definition does in general not allow us to easily compute the values for f_{ij}^r . Fortunately they can be found by solving a system of linear equations. To this end fix a state $j \in \mathcal{S}$ and define for each state $i \in \mathcal{S}$ a corresponding variable x_i (representing f_{ij}^r). It can be shown, in a similar way as for the hitting probabilities in the previous subsection, that the following result holds:

The probability-weighted averages of the cumulative rewards f_{ij}^r
form the least non-negative solution to the following
system of equations: $x_i = r(i) \cdot f_{ij}^r + \sum_{k \in \mathcal{S} \setminus \{j\}} P_{ik} x_k$

From (B.38) it immediately follows that $f_{ij}^r = 0$ in case state j is not accessible from state i . Hence the corresponding variable x_i can be set to 0 in this case.

The number of states in a path for which the reward is counted in (B.38) is equal to the number of transitions of that path. By assigning a reward value 1 to each state, we can thus also compute the *expected number of steps* required to hit a state. This is illustrated in the following example.

Example B.12 (Expected cumulative reward - hitting a state). In Example B.11 we computed f_{16} , i.e. the probability that the chain in Figure B.5 ever hits state 6, starting from state 1. In this example we will compute the expected time $\frac{f_{16}^r}{f_{16}}$ required to do so. We will compute f_{16}^r by solving the equations in (B.39) and by assigning a reward value 1 to every state. We then obtain $x_1 = f_{16} + \frac{1}{2}x_2$, $x_2 = f_{26} + \frac{2}{3}x_2 + \frac{1}{3}x_3$ and $x_3 = f_{36} + \frac{1}{5}x_4 + \frac{2}{5}x_5$ (notice that variables x_4, x_5, x_6, x_7 and x_8 are not required to compute f_{16}^r). Solving yields $x_1 = \frac{15}{10}$, $x_2 = \frac{8}{5}$ and $x_3 = \frac{2}{5}$. Hence $f_{16}^r = \frac{15}{10}$ and $\frac{f_{16}^r}{f_{16}} = \frac{15}{7}$. The expected number of steps required to move from state 1 to state 6 is therefore $\frac{15}{7}$. It is illustrative to also compute f_{16}^r by applying (B.38) directly. An infinite number

of paths exist from state 1 to state 6 (and not visiting state 3 in between):

- path 1, 6 with probability $\frac{1}{2}$ and reward 1
- path 1, 2, 3, 6 with probability $\frac{1}{2} \cdot \frac{2}{3}^0 \cdot \frac{1}{3} \cdot \frac{2}{5}$ and reward 3
- path 1, 2, 2, 3, 6 with probability $\frac{1}{2} \cdot \frac{2}{3}^1 \cdot \frac{1}{3} \cdot \frac{2}{5}$ and reward 4
- path 1, 2, 2, 2, 3, 6 with probability $\frac{1}{2} \cdot \frac{2}{3}^2 \cdot \frac{1}{3} \cdot \frac{2}{5}$ and reward 5
- ...

Therefore $f_{16}^r = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{3} \cdot \frac{2}{5} \sum_{n=0}^{\infty} \frac{2^n}{3}(n+3) = \frac{1}{2} + \frac{1}{15} \sum_{n=0}^{\infty} \frac{2^n}{3}(n+3) = \frac{1}{2} + \frac{1}{15}(3 \sum_{n=0}^{\infty} \frac{2^n}{3} + \sum_{n=0}^{\infty} \frac{2^n}{3} \cdot n) = \frac{1}{2} + \frac{1}{15}(\frac{3}{1-\frac{2}{3}} + \sum_{n=0}^{\infty} \frac{2^n}{3} \cdot n) = \frac{1}{2} + \frac{1}{15}(9 + \sum_{n=0}^{\infty} \frac{2^n}{3} \cdot n) = {}^a \frac{1}{2} + \frac{1}{15}(9 + \frac{\frac{2}{3}}{(1-\frac{2}{3})^2}) = \frac{1}{2} + \frac{1}{15}(9 + 6) = \frac{15}{10}$. Computing f_{16}^r by solving the equations is clearly more convenient.

^aFor $|x| < 1$, $\sum_{n=0}^{\infty} n \cdot x^n = \frac{x}{(1-x)^2}$.

Exercise B.24 (Hiccups in a video application). A video application displays a streaming movie with a rate of 60 frames per second. Upon a clock event, a movie frame (480×640 pixels) is read and removed from a buffer (if the buffer is non-empty) and is being displayed. The buffer can store up to 3 frames. Between two clock events new frames arrive from a network with probability p ($0 < p < 1$). In that case the buffer has completely refilled when the next clock event is raised. If the buffer is empty at the moment the clock event raised, a hiccup occurs.

- (a) Model this video application as a Markov chain. Explain the time domain, the states and the transitions chosen.
- (b) Compute (possibly making use of symbolic solver such as Mathematica) the expected number of video frames that are displayed (as function of p) before a hiccup occurs, when the video application starts with a full buffer.
- (c) Estimate (by using a solver) the values of p for which the expected time before a hiccup occurs (starting from a full buffer) is at least an hour.

See answer.

B.5.3 Probability to hit a set of states

For long-run analysis, which we discuss in the next section, we will require the hit probabilities of sets of states, typically of recurrent classes of states. For set $H \subseteq \mathcal{S}$ we first define h_{iH} as the probability that one of the states in H is ever visited, starting from state i . h_{iH} is sum of the probabilities of all paths from i to H with a length of at least zero, and not

visiting state j in between:

$$h_{iH} = \sum \begin{cases} \{P(i, i_1, \dots, i_n) \mid i, i_1, \dots, i_n \text{ is a path of length } n \geq 0 \\ \text{such that } i \in H \text{ implies } n = 0, \text{ and } i \notin H \text{ implies} \\ n \geq 1, i_n \in H \text{ and } i_k \notin H \text{ for all } k = 1 \dots n-1\} \end{cases} \quad (\text{B.40})$$

Again, the hitting probabilities h_{iH} can be computed by solving a set of linear equations. Define for each state $i \in \mathcal{S}$ a corresponding variable x_i (representing the hit probability h_{iH}) and consider the system of linear equations defined by

$$x_i = \begin{cases} 1 & \text{if } i \in H \\ \sum_{k \in \mathcal{S}} P_{ik} x_k & \text{if } i \notin H \end{cases}$$

It can be shown (in a similar way as for the hitting probabilities f_{ij}) that this set of equations is solved by substituting for each variable x_i the value h_{iH} . Again, multiple solutions can exist, but the hitting probabilities form the least non-negative solution.

The hitting probabilities h_{iH} form the least non-negative solution to the following set of equations

$$x_i = \begin{cases} 1 & \text{if } i \in H \\ \sum_{k \in \mathcal{S}} P_{ik} x_k & \text{if } i \notin H \end{cases} \quad (\text{B.41})$$

In practice the system of equations can be efficiently solved by realizing that $x_i = 0$ in case no state in H is accessible from i . Further notice that $x_i = 1$ in case $i \in H$.

Example B.13 (Hitting probabilities - hitting a set). Consider again the transition diagrams in Figure B.5. Assume we like to compute the hitting probabilities for class $C_5 = \{6, 7, 8\}$. Starting in the states of this class, we obtain $x_6 = x_7 = x_8 = 1$. For state 3 we obtain $x_3 = \frac{2}{5}x_6 = \frac{2}{5}$. For state 2 we get $x_2 = \frac{2}{3}x_2 + \frac{1}{3}x_3 = \frac{2}{3}x_2 + \frac{2}{15}$. Hence $x_2 = \frac{2}{5}$. For state 1 the equations are $x_1 = \frac{1}{2}x_2 + \frac{1}{2}x_6 = \frac{1}{5} + \frac{1}{2} = \frac{7}{10}$. So $h_{1C_5} = f_{16}$ (see Example B.11) in this case since the only way to enter C_5 is through state 6. To illustrate the computation via paths consider h_{6C_5} . Since state 6 is already in C_5 only paths of length 0 (starting in state 6) should be considered (according to (B.40)). Only one such path exists which is 6 and its probability is 1 (see (B.18)), consistent with the computation of x_6 above.

Now that we have defined two notions of hitting probabilities, it is time to compare them. It may be tempting to think that the only difference concerns hitting a state versus hitting a set of states, but this false. To understand the difference consider f_{ii} versus $h_{i\{i\}}$. Now $h_{i\{i\}}$ always equals 1 since the chain hits set $\{i\}$ already at the start (after 0 steps). f_{ii} denotes the probability that the chain eventually *returns* to i after it has just left i (i.e. after 1 or more steps), and in general does not equal 1. However in case $i \neq j$ we do have that $f_{ij} = h_{i\{j\}}$. It may further be tempting to think that in case $i \notin H$, $h_{iH} = \sum_{j \in H} f_{ij}$. This is false however! $\sum_{j \in H} f_{ij}$ counts probabilities multiple times and can deliver a value that exceeds 1.

Example B.14 (Gambler's ruin - win probability). Consider the example of the gambler's ruin again, depicted in Figure B.3. Assume we want to compute the probability that the gambler wins the game, starting with an initial capital of €100. We can do so by computing h_{2H} , the probability that recurrent class $H = \{4\}$ is ever hit from state 2. The equations are $x_1 = x_1$, $x_2 = \frac{1}{2}x_1 + \frac{1}{2}x_3$, $x_3 = \frac{1}{2}x_2 + \frac{1}{2}x_4$, $x_4 = 1$. The equation $x_1 = x_1$ has more than one solution, but the least non-negative solution is $x_1 = 0$ (which also immediately follows from the fact that H cannot be accessed from state 1). Solving the equations yields $x_1 = 0$, $x_2 = \frac{1}{3}$, $x_3 = \frac{2}{3}$ and $x_4 = 1$ and hence $h_{1H} = 0$, $h_{2H} = \frac{1}{3}$, $h_{3H} = \frac{2}{3}$ and $h_{4H} = 1$. Observe that the answer $h_{2H} = \frac{1}{3}$ closely resembles $P_{2,4}^{10} \approx 0.333$ (see Example B.6). In the next section we will learn that indeed $P_{2,4}^n \rightarrow \frac{1}{3}$ for $n \rightarrow \infty$.

Exercise B.25 (Hitting a state versus hitting a singleton state set). Show that for all $i, j \in \mathcal{S}$ with $i \neq j$, $f_{ij} = h_{i\{j\}}$, based on equations (B.36) and (B.41).
See answer.

B.5.4 Expected cumulative reward until hitting a set of states

We will define the expected cumulative reward earned until some state in (a set of states) H is hit, starting from state i , as the probability-weighted average of the cumulative rewards of the paths from i to H (not visiting j in between and not counting the reward obtained in the target state in H), normalized to the probability that H is actually hit.

The expected cumulative reward earned until a state in H is hit, starting from state i , is defined as $\frac{h_{iH}^r}{h_{iH}}$ (B.42)

Here h_{iH} is the hitting probability defined in the previous subsection and h_{iH}^r is the probability-weighted average defined in terms of the paths from i to H :

$$h_{iH}^r = \sum_{\substack{i, i_1, \dots, i_n \text{ is a path of length } n \geq 1 \text{ such that} \\ i \notin H, i_n \in H \text{ and } i_k \notin H \text{ for all } k = 1, \dots, n-1}} \{P(i, i_1, \dots, i_n) \cdot (r(i) + r(i_1) + \dots + r(i_{n-1})) \mid$$
(B.43)

Note that for a path of length 0, state i must necessarily be in H , resulting in zero reward (since the reward of the final state is not counted). Therefore, in contrast to (B.40) we only have to consider paths of positive length. Note that indeed $h_{iH}^r = 0$ when $i \in H$ since the set of paths is empty in that case.

The values h_{iH}^r can be computed by solving a system of linear equations. If we fix state set $H \subseteq \mathcal{S}$ and define for each state $i \in \mathcal{S}$ a corresponding variable x_i (representing h_{iH}^r) we can show that the following result holds:

The probability-weighted averages of the cumulative rewards h_{iH}^r form the least non-negative solution to the following set of equations (B.44)

$$x_i = \begin{cases} 0 & \text{if } i \in H \\ r(i) \cdot h_{iH} + \sum_{k \in S} P_{ik} x_k & \text{if } i \notin H \end{cases}$$

From (B.43) it immediately follows that $h_{iH}^r = 0$ in case no state in H is accessible from state i . Hence the corresponding variable x_i can be set to 0 in this case.

In the same way as we compute the expected number of steps required to hit a state, we can compute the expected number of steps to hit a set of states, by defining a reward that assigns a value 1 to each state.

Example B.15 (Gambler's ruin - expected number of spins until win). Consider the example of the gambler's ruin again, depicted in Figure B.3. Assume we want to compute the expected number of spins required before the gambler wins the game, starting with an initial capital of €100. We take $H = \{4\}$ and define a new reward function r' assigning the value 1 to each state. We have to compute $\frac{h_{2H}^{r'}}{h_{2H}}$. The hitting probability probabilities are already computed in Example B.14 as $h_{1H} = 0, h_{2H} = \frac{1}{3}, h_{3H} = \frac{2}{3}$ and $h_{4H} = 1$. To compute $h_{2H}^{r'}$ we use the equations $x_1 = 0, x_2 = \frac{1}{3} + \frac{1}{2}x_3, x_3 = \frac{2}{3} + \frac{1}{2}x_2$ and $x_4 = 0$. Hence $h_{2H}^{r'} = \frac{8}{9}$ and thus $\frac{h_{2H}^{r'}}{h_{2H}} = 2\frac{2}{3}$. Thus if we would measure the required number of spins for a large number of games that are actually won by the player, the average of these numbers would be close to $2\frac{2}{3}$.

Exercise B.26 (Hitting recurrent states with probability 1). By definition, a state i is recurrent if $f_{ii} = 1$ (see (B.24)). When i is in class C , this implies that $f_{jj} = 1$ for each $j \in C$ (B.27). Show that the following generalized property also holds: if C is a recurrent class, then for all $i, j \in C$, $f_{ij} = 1$. Hint: use the system of equations in (B.36).

See answer.

Exercise B.27 (Rover in a maze). An autonomous rover tries to find its way of a maze consisting of 16 cells, see Figure B.8. The rover starts at the right-bottom cell, driving in the upward direction as indicated with the arrow. It can pass the dashed separation lines between cells, but not the solid lines (which represent walls). When arriving in a cell, it decides what cell it will visit next. It can choose from all neighboring cells, except for the cell it just arrived from. From the cells it can choose, it makes a decision with equal probabilities. So if it can go in three directions, each of these directions can be chosen with probability $\frac{1}{3}$. The goal of the rover is to find the exit cell (see figure), but along the way it might get stuck since it cannot turn around.

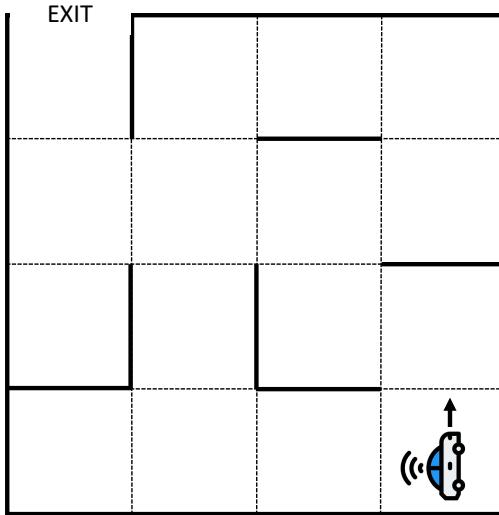


Figure B.8: An autonomous rover in a maze

- (a) What is the probability that the rover finds its way out of the maze?
- (b) What is the expected number of cells the rover visits until it finds the exit cell?
- (c) What is the expected number of times the rover visits the initial cell, before it finds the exit cell?

See answer.

B.6 Long-run analysis

In Section B.3 we learned how to perform transient analysis by computing $\pi^{(n)}$ and $E(r(X_n))$ for fixed values of n . Based on the Markov chain type (Section B.4) and hitting probabilities (Section B.5) we will study long-run behaviour when n grows very large.

B.6.1 Ergodic unichains

We will study the long-run behaviour step-by-step, starting with the following important fact:

The limiting distribution $\lim_{n \rightarrow \infty} \pi^{(n)} = \pi^{(\infty)}$ exists for every ergodic unichain (B.45)

This means that the Markov chains X_0, X_1, \dots converges to a random variable with probability distribution $\pi^{(\infty)}$. This mode of convergence is called *converge in distribution* since the sequence of distribution vectors converges to a limit distribution. Based on this fact we

can compute the long-run expected rewards as:

$$\lim_{n \rightarrow \infty} E(r(X_n)) = \lim_{n \rightarrow \infty} \pi^{(n)} r^T = \pi^{(\infty)} r^T$$

So how can we compute $\pi^{(\infty)}$? To answer this question recall equation (B.14) stating that $\pi^{(n+1)} = \pi^{(n)} P$. From this equation it follows that

$$\lim_{n \rightarrow \infty} \pi^{(n+1)} = \lim_{n \rightarrow \infty} \pi^{(n)} P$$

and thus that

$$\pi^{(\infty)} = \pi^{(\infty)} P$$

Since $\sum_{i \in \mathcal{S}} \pi_i^{(\infty)} = 1$ we therefore have that π^∞ is a solution to the system of equations given by $\pi = \pi P, \sum_{i \in \mathcal{S}} \pi_i = 1$. These equations are called the *balance equations* of a Markov chain and a solution to this system is called a *stationary distribution*.

The limiting distribution $\pi^{(\infty)}$ of an ergodic unichain is a solution to the balance equations $\pi = \pi P, \sum_{i \in \mathcal{S}} \pi_i = 1$ (B.46)

In general the balance equations can have multiple solutions, however:

The balance equations of an ergodic unichain have a unique solution (B.47)

Hence any ergodic unichain has a unique limiting distribution $\pi^{(\infty)}$ that can be found by solving the balance equations. But there is another important fact we can derive. From (B.15) we know that $\pi^{(n)} = \pi^{(0)} P^n$ and thus

$$\pi^{(\infty)} = \lim_{n \rightarrow \infty} \pi^{(n)} = \pi^{(0)} \lim_{n \rightarrow \infty} P^n = \pi^{(0)} P^\infty$$

Since $\pi^{(\infty)}$ exists, P^∞ exists as well. Moreover the limiting distribution is *independent* of the initial distribution $\pi^{(0)}$. As a consequence:

- $\pi^{(\infty)} = [1, 0, 0 \dots, 0] P^\infty$ and thus the first row of P^∞ equals $\pi^{(\infty)}$
- $\pi^{(\infty)} = [0, 1, 0 \dots, 0] P^\infty$ and thus the second row of P^∞ equals $\pi^{(\infty)}$
- $\pi^{(\infty)} = [0, 0, 1 \dots, 0] P^\infty$ and thus the third row of P^∞ equals $\pi^{(\infty)}$
- ...

We thus obtain the following conclusion

The limiting matrix P^∞ exists for any ergodic unichain, where each row of P^∞ equals $\pi^{(\infty)}$ (B.48)

The different entries in this matrix are interpreted as long-run transition probabilities. For each $i, j \in \mathcal{S}$, P_{ij}^∞ is the probability that the chain, starting from state i , will be in state j

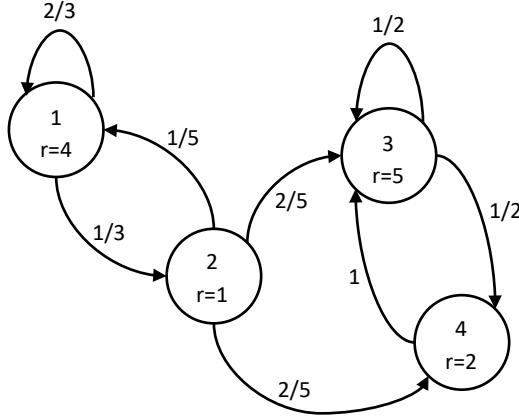


Figure B.9: An ergodic unichain with rewards

in the long run. Before we will give an example of ergodic unichains, we finalize with the following remark:

For an ergodic unichain, the limiting distribution $\pi^{(\infty)}$ is computed by $\pi^{(0)}P^\infty$ and the long-run expected reward $\lim_{n \rightarrow \infty} E(r(X_n))$ is computed by $\pi^{(\infty)}r^T$ (B.49)

Example B.16 (Limiting matrix ergodic unichain). In Figure B.9 a transition diagram of an ergodic unichain is drawn. The probability matrix is given by

$$P = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{3} & 0 & \frac{2}{5} & \frac{2}{5} \\ \frac{1}{5} & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The balance equations are

$$\begin{aligned} \pi_1 &= \frac{2}{3}\pi_1 + \frac{1}{5}\pi_2 \\ \pi_2 &= \frac{1}{3}\pi_1 \\ \pi_3 &= \frac{2}{5}\pi_2 + \frac{1}{2}\pi_3 + \pi_4 \\ \pi_4 &= \frac{2}{5}\pi_2 + \frac{1}{2}\pi_3 \\ \pi_1 + \pi_2 + \pi_3 + \pi_4 &= 1 \end{aligned}$$

Solving these equations gives $\pi_1 = 0$, $\pi_2 = 0$, $\pi_3 = \frac{2}{3}$, $\pi_4 = \frac{1}{3}$ and thus $\pi^{(\infty)} = [0, 0, \frac{2}{3}, \frac{1}{3}]$. Hence no matter the starting distribution, in the long run the chain will be in recurrent state 3 with probability $\frac{2}{3}$ and in recurrent state 4 with probability $\frac{1}{3}$. The limiting probabilities for the transient states 1 and 2 are both 0. The limiting matrix is given

by

$$P^\infty = \begin{bmatrix} 0 & 0 & \frac{2}{3} & \frac{1}{3} \\ 0 & 0 & \frac{2}{3} & \frac{1}{3} \\ 0 & 0 & \frac{2}{3} & \frac{1}{3} \\ 0 & 0 & \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

The elements in this matrix denote long-run transition probabilities. $P_{13}^\infty = \frac{2}{3}$, for instance, denotes the probability that the chain, starting from state 1, will be in state 3 in the long run. Based on the reward vector $r = [4, 1, 5, 2]$ and $\pi^{(\infty)}$ we can compute the long-run expected reward as $\pi^{(\infty)} r^T = [0, 0, \frac{2}{3}, \frac{1}{3}] [4, 1, 5, 2]^T = \frac{2}{3} \cdot 5 + \frac{1}{3} \cdot 2 = 4$. Hence in the long-run, we expect to obtain 4 units of reward after each transition.

Exercise B.28 (Limiting matrix ergodic unichain). Consider the Markov chain

of Exercise B.21 again having transition probability matrix $P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}$.

- (a) Approximate P^5 , P^{10} and P^{15}
- (b) Compute P^∞

See answer.

Exercise B.29 (Video application - limiting distribution). Consider the Markov chain of the video application in Exercise B.24.

- (a) Compute (e.g. using a symbolic solver) the probability that the buffer contains two frames in the long-run.
- (b) For what value of p is this probability maximal? What is the maximal probability?

See answer.

B.6.2 Non-ergodic unichains

As a next step we consider non-ergodic unichains in general. These chains have a single recurrent periodic class, the states of which are visited in periodic patterns, dependent on the time of entry of the class. Therefore the limit $\lim_{n \rightarrow \infty} P^n$ does not exist:

For a unichain $\lim_{n \rightarrow \infty} P^n$ exists
if and only if its recurrent class is aperiodic (B.50)

This implies that (in general⁷) neither $\lim_{n \rightarrow \infty} \pi^{(n)}$ nor $\lim_{n \rightarrow \infty} \pi^{(n)} r^T$ exists for periodic unichains. However, the balance equations still have a unique solution. Even stronger:

The balance equation have a unique stationary distribution
if and only if the chain is a unichain (B.51)

But what does the stationary distribution represent? We claim that it equals the so-called *Cezàro* limit of $\pi^{(n)}$ defined by

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)}$$

To see this, recall that $\pi^{(k+1)} = \pi^{(k)} P$ for all $k \geq 0$. By taking the summation we then get

$$\frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k+1)} = \left(\frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)} \right) P$$

and by taking the limit we thus obtain

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k+1)} = \lim_{n \rightarrow \infty} \left(\frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)} \right) P$$

Now $\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k+1)} = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=1}^n \pi^{(k)} = \lim_{n \rightarrow \infty} \frac{1}{n} (\pi^{(n)} - \pi^{(0)}) + \lim_{m \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)}$
 $= \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)}$ and thus

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)} = \lim_{n \rightarrow \infty} \left(\frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)} \right) P$$

The *Cezàro* limit $\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)}$ is therefore a solution to the balance equations. This solution is called the *Cezàro limiting distribution* of $\pi^{(n)}$ which we will denote by $\pi^{(\infty)}$. It can be shown that this limit always exists for finite-state Markov chains.

The unique solution to the balance equations of a unichain
equals the Cezàro limiting distribution $\pi^{(\infty)}$ (B.52)

So obviously in case of an ergodic unichain $\pi^{(\infty)} = \pi^{(\infty)}$.

Just as $\pi^{(\infty)} = \pi^{(0)} P^\infty$ for ergodic unichains, we can derive that $\pi^{(\infty)} = \pi^{(0)} P^\infty$ for unichains in general, where P^∞ is the Cezàro limit of P^n defined by $\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} P^k$:

$$\pi^{(\infty)} = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)} = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(0)} P^k = \pi^{(0)} \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} P^k = \pi^{(0)} P^\infty$$

⁷These limits may still exist for specific initial distributions or reward vectors.

It can be shown that P^∞ exists for every finite-state Markov chain. It further follows that the Cezàro limiting distribution is *independent* of the initial distribution $\pi^{(0)}$. Therefore the Cezàro limiting matrix has the same shape as the limiting matrix for an ergodic unichain:

The Cezàro limiting matrix P^∞ exists for any unichain,
where each row of P^∞ equals $\pi^{(\infty)}$
and $\pi^{(\infty)} = \pi^{(0)} P^\infty$

Notice that for an ergodic unichain $P^\infty = P^\infty$.

We know that for ergodic unichains, the long-run expected reward is computed by $\pi^{(\infty)} r^T$. For unichains in general $\pi^{(\infty)}$ does not exist, but we can use $\pi^{(\infty)}$ instead to obtain

$$\pi^{(\infty)} r^T = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} \pi^{(k)} r^T = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} E(r(X_k)) = {}^8 \lim_{n \rightarrow \infty} E\left(\frac{1}{n} \sum_{k=0}^{n-1} r(X_k)\right)$$

Hence $\pi^{(\infty)} r^T$ represents the long-run expected *average* reward. For an ergodic unichain $\pi^{(\infty)} r^T = \pi^{(\infty)} r^T$ and thus the long-run expected reward equals the long-run expected average reward in this case.

The long-run expected average reward $\lim_{n \rightarrow \infty} E\left(\frac{1}{n} \sum_{k=0}^{n-1} r(X_k)\right)$
exists for any unichain and equals $\pi^{(\infty)} r^T$

Until this point we did not yet explain what $\pi^{(\infty)}$ and P^∞ actually mean. In fact they represent long-run expected fractions of time the chain spends in its states. To see this, assume for instance that $r = [1, 0, \dots, 0]$. Then by (B.54), $\pi^{(\infty)} r^T$ equals the long-run expected fraction of time the chain spends in state 1. But $\pi^{(\infty)} r^T = \pi_1^{(\infty)}$, so $\pi_1^{(\infty)}$ has the same interpretation. Further if we set $\pi^{(0)} = [1, 0, \dots, 0]$ it follows from (B.53) that $\pi_1^{(\infty)} = P_{11}^\infty$. Thus P_{11}^∞ represents the long-run expected fraction of time the chain spends in state 1, given that it started in state 1.

Example B.17 (Cezàro limiting matrix non-ergodic unichain). Figure B.10 shows a transition diagram of a non-ergodic unichain with reward r . The probability matrix is given by

$$P = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} & 0 & 0 \\ \frac{3}{5} & 0 & \frac{2}{5} & \frac{2}{5} \\ \frac{1}{5} & 0 & \frac{2}{5} & \frac{2}{5} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

⁸The expectation operator is linear in case of a *finite sum*, implying that the expected value of a finite sum of random variables is equal to the sum of their individual expected values, regardless of whether these variables are independent.

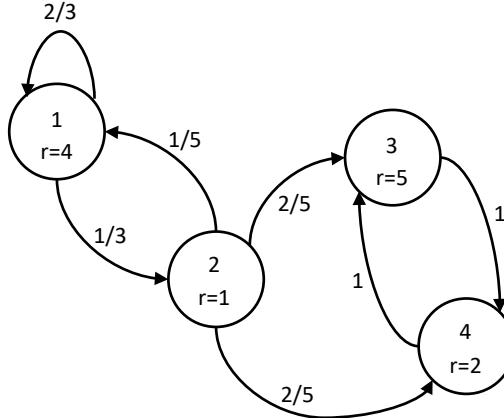


Figure B.10: A unichain with rewards

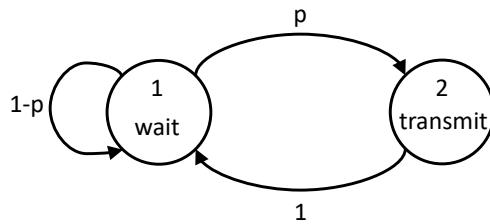


Figure B.11: Transition diagram of packet generator

When we compute the (incomplete) first powers of this matrix we obtain

$$P^2 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 0 \\ \cdot & \cdot & 0 & 1 \end{bmatrix}, P^3 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 0 & 1 \\ \cdot & \cdot & 1 & 0 \end{bmatrix}, P^4 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 0 \\ \cdot & \cdot & 0 & 1 \end{bmatrix}$$

So indeed, the limit $\lim_{n \rightarrow \infty} P^n$ does not exist, and neither does $\lim_{n \rightarrow \infty} \pi^{(n)}$. The balance equations, however, still have a unique solution which is given by $\pi_1 = 0, \pi_2 = 0, \pi_3 = \frac{1}{2}, \pi_4 = \frac{1}{2}$. Therefore $\pi^{(\infty)} = [0, 0, \frac{1}{2}, \frac{1}{2}]$ and

$$P^\infty = \begin{bmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Hence no matter the starting distribution, the long-run expected fraction of time the chain spends in states 3 and 4 are both $\frac{1}{2}$. The long-run expected fraction of time spent in states 1 or 2 is 0. The long-run expected average reward is given by $\pi^{(\infty)} r^T = [0, 0, \frac{1}{2}, \frac{1}{2}] [4, 1, 5, 2]^T = \frac{1}{2} \cdot 5 + \frac{1}{2} \cdot 2 = 3\frac{1}{2}$.

Exercise B.30 (Packet generator - generated load). A packet generator for a slotted communication medium behaves as a Markov chain with two states. The transition diagram is shown in Figure B.11. State 1 represents a wait state and state 2 represents a transmit state. A packet is produced only when the generator is in the transmit state. p ($0 \leq p \leq 1$) is a parameter to control the generated load, i.e. the expected number of packets produced per time slot.

- (a) Give an expression of the generated load in terms of p .
- (b) What are the minimal and the maximal loads that can be generated?

See answer.

Exercise B.31 (Expected fraction of time spent in a state equals reciprocal of expected return time). One can show that for any state in a recurrent class, the long-run expected fraction of time the chain spends in this state equals the reciprocal of the expected return time to that state. Consider the Markov chain with transition probability matrix

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{5}{6} & 0 & \frac{1}{6} \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

- (a) Show that this chain is irreducible (implying that it consists of single recurrent class), but not ergodic.
- (b) Show that the above property holds for state 1 of this chain.

See answer.

B.6.3 General Markov chains

Now that we have understood unichains, we will look into the long-run behaviour of Markov chains in general. We already know from (B.51) that a unique solution to the balance equations exists if and only if the chain is a unichain. In case the chain has more than one recurrent class, the balance equations have multiple solutions. In that case limiting distributions still exist, but are dependent on the initial distribution and the limiting matrices. In general the following results hold:

The Cezàro limiting matrix P^∞ exists for any Markov chain
and so does the Cezàro limiting distribution $\pi^{(\infty)}$
which is computed by $\pi^{(0)}P^\infty$ (B.55)

The long-run expected average reward is computed by $\pi^{(\infty)}r^T$

We will show how to compute P^∞ . For each $i, j \in \mathcal{S}$, P_{ij}^∞ is computed as follows:

$$P_{ij}^\infty = \begin{cases} h_{iC} \cdot \pi_j^C & \text{if } j \text{ is in recurrent class } C \text{ and } \pi_j^C \text{ is the solution} \\ & \text{for state } j \text{ of the balance equations for } C \\ 0 & \text{if } j \text{ is a transient state} \end{cases} \quad (\text{B.56})$$

Hence to compute the limiting matrices we look at the target state j . In case this target state is transient it will not be visited in the long-run (irrespective of the starting state) and therefore $P_{ij}^\infty = 0$. Otherwise state j is in some recurrent class C . Starting from i , the chain will end up in this class with hitting probability h_{iC} and then remain in that class. When arrived in that class, the long-run average fraction of time spent in state j equals the (unique) solution π_j^C for state j of the balance equations for C . Therefore $P_{ij}^\infty = \pi_j^C \cdot h_{iC}$. Notice that when $i \in C$, $h_{iC} = 1$ and thus $P_{ij}^\infty = \pi_j^C$ in that case. In case i is in a recurrent class different from C , $h_{iC} = 0$ and thus $P_{ij}^\infty = 0$.

We thus have that the Cezàro limits always exist. This is not true for the normal limits. These normal limits only exist in case of an ergodic Markov chain. In that case the Cezàro limits and the normal limits are identical:

The normal limits $P^\infty, \pi^{(\infty)}$ and $\pi^{(\infty)r^T}$ exist
if and only if the chain is ergodic

If these limits exist, they identical to their Cezàro counterparts

The limiting properties of the different Markov chain types are summarized in Table B.2.

Example B.18 (Computing limiting matrix). Consider the transition diagrams in Figure B.5. Since class C_4 is periodic P^∞ does not exist. So we will compute P^∞ . States 1, 2 and 3 are transient states and thus the first three columns of P^∞ contain only zeros. Solving the balance equations for class C_4 gives $\pi_4^{C_4} = \pi_5^{C_4} = \frac{1}{2}$. These probabilities determine rows 4 and 5 of P^∞ . For class C_5 we obtain $\pi_6^{C_5} = \frac{4}{9}$, $\pi_7^{C_5} = \frac{2}{9}$ and $\pi_8^{C_5} = \frac{1}{3}$, fixing rows 6, 7 and 8. For the transient states, we compute the probabilities to hit classes C_4 and C_5 . For C_5 we obtain (see also Example B.13), $h_{1C_5} = \frac{7}{10}$, $h_{2C_5} = \frac{2}{5}$ and $h_{3C_5} = \frac{2}{5}$ and for class C_4 we obtain $h_{1C_4} = \frac{3}{10}$, $h_{2C_4} = \frac{3}{5}$ and $h_{3C_4} = \frac{3}{5}$. We can then determine the values in rows 1, 2 and 3. For instance $P_{16}^\infty = h_{1C_5} \cdot \pi_6^{C_5} = \frac{7}{10} \cdot \frac{4}{9} = \frac{14}{45}$ and $P_{24}^\infty = h_{2C_4} \cdot \pi_4^{C_4} = \frac{3}{5} \cdot \frac{1}{2} = \frac{3}{10}$. The Cezàro limiting matrix is then as follows

| Markov chain type | unichain | non-unichain |
|-------------------|---|--|
| ergodic | <ul style="list-style-type: none"> (i) normal limits and Cezàro limits exist and are identical (ii) $\pi^{(\infty)}r^T$ is long-run expected reward (iii) $\pi^{(\infty)}$ is unique solution of balance equations; $\pi^{(\infty)}$ independent of $\pi^{(0)}$ (iv) all rows in P^∞ are identical to $\pi^{(\infty)}$ | <ul style="list-style-type: none"> (i) normal limits and Cezàro limits exist and are identical (ii) $\pi^{(\infty)}r^T$ is long-run expected reward (iii) balance equations have multiple solutions; $\pi^{(\infty)}$ depends on $\pi^{(0)}$ (iv) rows in P^∞ are not all equal |
| non-ergodic | <ul style="list-style-type: none"> (i) only Cezàro limits exist (ii) $\pi^{(\infty)}r^T$ is long-run expected average reward (iii) $\pi^{(\infty)}$ is unique solution of balance equations; $\pi^{(\infty)}$ independent of $\pi^{(0)}$ (iv) all rows in P^∞ are identical to $\pi^{(\infty)}$ | <ul style="list-style-type: none"> (i) only Cezàro limits exist (ii) $\pi^{(\infty)}r^T$ is long-run expected average reward (iii) balance equations have multiple solutions; $\pi^{(\infty)}$ depends on $\pi^{(0)}$ (iv) rows in P^∞ are not all equal |

Table B.2: Markov chain types and their limiting properties

$$P^\infty = \begin{bmatrix} 0 & 0 & 0 & \frac{3}{20} & \frac{3}{20} & \frac{14}{45} & \frac{7}{45} & \frac{7}{30} \\ 0 & 0 & 0 & \frac{3}{10} & \frac{3}{10} & \frac{45}{8} & \frac{45}{4} & \frac{15}{2} \\ 0 & 0 & 0 & \frac{3}{10} & \frac{3}{10} & \frac{45}{45} & \frac{45}{45} & \frac{15}{15} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{4}{9} & \frac{2}{9} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & \frac{4}{9} & \frac{2}{9} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & \frac{4}{9} & \frac{2}{9} & \frac{1}{3} \end{bmatrix}$$

Based on this matrix we can compute the Cezáro limiting distribution for any initial distribution. For instance if $\pi^{(0)} = [\frac{1}{4}, \frac{3}{4}, 0, 0, 0, 0, 0, 0]$ we get $\pi^{(0)}P^\infty = [0, 0, 0, \frac{21}{80}, \frac{21}{80}, \frac{19}{90}, \frac{19}{180}, \frac{19}{120}]$.

Example B.19 (Gambler's ruin continued). Consider the example of the gambler's ruin again, depicted in Figure B.3. Both recurrent classes $\{1\}$ and $\{4\}$ are aperiodic and hence P^∞ exists. Solving the balance equations for these classes yields $\pi_1^{\{1\}} = 1$ and $\pi_4^{\{4\}} = 1$. These determine rows 1 and 4 of P^∞ . States 2 and 3 are transient, implying that columns 2 and 3 of P^∞ contain only zeroes. We still have to determine $P_{21}^\infty, P_{24}^\infty, P_{31}^\infty$ and P_{34}^∞ . Now $P_{21}^\infty = h_{2\{1\}}\pi_1^{\{1\}} = \frac{2}{3}$, $P_{24}^\infty = h_{2\{4\}}\pi_4^{\{4\}} = \frac{1}{3}$, $P_{31}^\infty = h_{3\{1\}}\pi_1^{\{1\}} = \frac{1}{3}$ and $P_{34}^\infty = h_{3\{4\}}\pi_4^{\{4\}} = \frac{2}{3}$. Therefore

$$P^\infty = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{2}{3} & 0 & 0 & \frac{1}{3} \\ \frac{1}{3} & 0 & 0 & \frac{2}{3} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that the matrix P^{10} in Example B.5 already approximates this matrix closely. Based on P^∞ , the limiting distributions for every initial distribution can be computed. For instance if $\pi^{(0)} = [0, \frac{1}{4}, \frac{3}{4}, 0]$, $\pi^{(\infty)} = \pi^{(0)}P^\infty = [\frac{5}{12}, 0, 0, \frac{7}{12}]$. The corresponding long-run expected amount of cash is $[\frac{5}{12}, 0, 0, \frac{7}{12}][0, 100, 200, 300]^T = €175$.

Exercise B.32 (Composition of two parallel packet generators - generated load). Two packet generators from Exercise B.30 are offering packets to a time-slotted communication medium, both using configuration parameter p . When both generators offer a packet in the same time slot, a collision occurs resulting in the loss of both packets. When precisely one generator offers a packet, this packet is successfully transmitted. In case no packet is offered, the time-slot is left unused.

- (a) Model this system as a Markov chain. Explain the meaning of the chosen states and transitions.
- (b) Compute the utilization of the medium.

- (c) For what setting of the configuration parameter p do we obtain optimal utilization. What is the optimal utilization?

See answer.

Exercise B.33 (Computer system - throughput). A computer system executes tasks of three different types, A , B and C . After the execution of a task of type A , such a task is executed again with probability $1 - p$ and with probability p this is a task of type B . After the execution of a type B task, a type C task follows with probability p and a type A task with probability $1 - p$. The execution of a task of type C is always followed by a task of type A .

- (a) Construct a Markov chain that models the behaviour of the computer system.
- (b) Let π_X denote the probability that the system is in state X in the long-run. Compute π_A , π_B and π_C .
- (c) The execution of tasks of types A , B and C take respectively 1, $3 + 3p$ and $2p$ seconds. Give an expression of the throughput of this computer system, i.e. the number of tasks executed per second in the long-run.
- (d) What is the minimal throughput of the computer system?

See answer.

B.7 Discrete-event simulation

Markov models are rather restricted in their primitives to model complex systems. Consider for instance the system consisting of two parallel packet generator in Exercise B.32. It would have been convenient to model this system as a parallel composition of two packet generators, instead of having to define a flat transition diagram as in Figure B.18. Constructs (such as the parallel composition construct) to conveniently model complex systems, are typically offered by tools that are used in performance modeling practice. The more primitive Markov models are still used in these tools, but then as underlying semantic models which are either implicitly or explicitly defined.

The state spaces of these underlying Markov chains can be very large or can even be infinite. While a single packet generator (in Exercise B.32) has only 2 states, the composition of two generators contains $2 \times 2 = 4$ states. If we would like to model a system with 100 packet generators, we would obtain a state space that is too large to be stored explicitly in a computer and to effectively compute transient or long-run performance properties. Therefore the tools that are used in performance modeling practice, often *estimate* performance properties instead of *computing* them.

Estimation of performance properties is performed by *discrete-event simulation*. In a discrete-event simulation one or more paths are generated through the Markov chain. For each path the simulator starts in some initial state. Upon the event of transitioning, a next state is chosen and visited. The choice should respect the transition probabilities and is made using a (pseudo)random number generator. This process repeats itself until a stop criterion is satisfied.

This section explains the basic theory of estimation based on confidence intervals. Estimations of expected rewards, expected rewards until hit and long-run expected average rewards are treated as special cases. For an example of a modeling language which incorporates this theory we refer to the Parallel Object-Oriented Specification Language (POOSL). POOSL comes with a scalable simulation tool [22] to handle industrial-sized problems.

B.7.1 Basic estimation theory

Consider a sequence of independent identically distributed random variables Y_0, Y_1, \dots all with the same expected value $E(Y_n) = \mu$ and variance $\text{Var}(Y_n) = \sigma^2$, where $\text{Var}(Y_n)$ is defined as $E((Y_n - \mu)^2)$, i.e. as the expected value of the squared deviation from the mean.

Our goal is to estimate the value of μ based on a simulation sequence y_0, y_1, \dots, y_{M-1} of *realizations* of the first M random variables. Thus y_n ($0 \leq n \leq M$) denotes a value that random variable Y_n can take. We name the estimation $\hat{\mu}$ and define it as

$$\hat{\mu} = \frac{1}{M} \sum_{n=0}^{M-1} y_n$$

The fact that $\hat{\mu}$ is an estimation of μ is based on the *strong law of large numbers*⁹:

Sequence $\frac{1}{M} \sum_{n=0}^{M-1} Y_n$ converges almost surely to μ ,
 where μ denotes the random variable
 that assumes value μ with probability 1

(B.58)

This means that the probability of the set of infinite sequences y_0, y_1, \dots of observations of Y_0, Y_1, \dots for which $\lim_{M \rightarrow \infty} \frac{1}{M} \sum_{n=0}^{M-1} y_n = \mu$ equals 1. Hence for any infinite sequence y_0, y_1, \dots we know *almost surely* that $\lim_{M \rightarrow \infty} \frac{1}{M} \sum_{n=0}^{M-1} y_n$ equals μ . But then $\hat{\mu} = \sum_{n=0}^{M-1} y_n$ approximates μ .

$\hat{\mu} = \frac{1}{M} \sum_{n=0}^{M-1} y_n$ is called a *point estimation* of μ . If we replace the realizations y_n with their corresponding random variables Y_n we obtain expression $\frac{1}{M} \sum_{n=0}^{M-1} Y_n$ which is called a *point estimator* of μ . Notice that the point estimator is a random variable of which a corresponding point estimation is a realization.

The absolute difference between μ and $\hat{\mu}$ is called the *absolute estimation error*. If we divide

⁹This law also requires $E(|Y_1|) < \infty$.

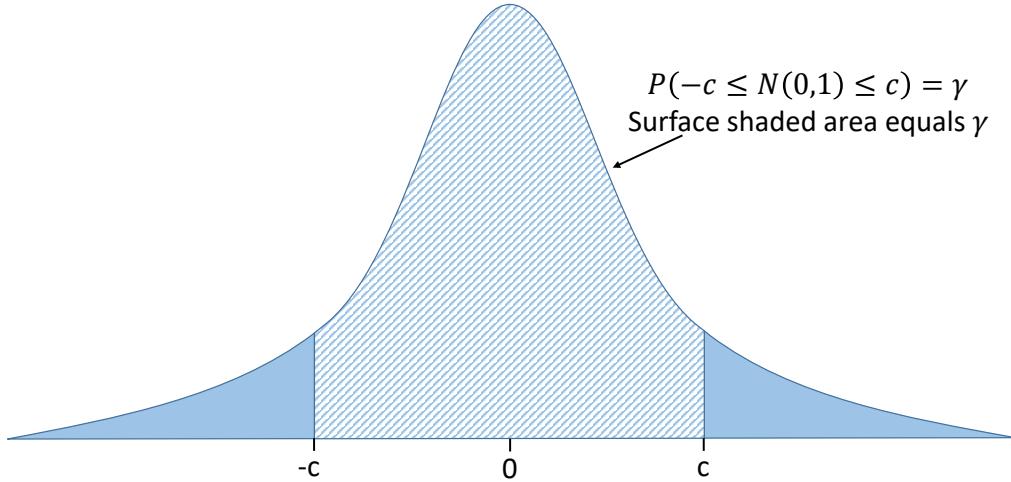


Figure B.12: Standard normal distribution

the absolute error by the norm of μ , we obtain the *relative estimation error*¹⁰. Hence

$$\begin{aligned} \text{The absolute estimation error is defined by } & |\mu - \hat{\mu}| \text{ and} \\ \text{the relative estimation error is defined by } & \frac{|\mu - \hat{\mu}|}{|\mu|} \end{aligned} \tag{B.59}$$

Notice that the estimation errors can be made as small as desired¹¹ by choosing M sufficiently large. So how long should the simulation sequence be so that we obtain acceptable errors?

The foundation to answer this question is the *central limit theorem*:

$$\begin{aligned} \text{Sequence } \sqrt{M} \frac{\frac{1}{M} \sum_{n=0}^{M-1} Y_n - \mu}{S_M} \text{ converges in distribution to } N(0, 1), \\ \text{where } S_M = \sqrt{\frac{1}{M} \sum_{n=0}^{M-1} (Y_n - \frac{1}{M} \sum_{m=0}^{M-1} Y_m)^2} \text{ and where} \\ N(0, 1) \text{ is a random variable with a standard normal distribution} \end{aligned} \tag{B.60}$$

The central limit theorem states that for sufficiently large M (as a practical rule of thumb one uses $M \geq 30$), the distribution of $\sum_{n=0}^M Y_n$ is approximately normal. Since the variables Y_n are independent and identically distributed (with mean μ and variance σ^2), we have $E(\sum_{n=0}^M Y_n) = M\mu$ and $Var(Y_n) = M\sigma^2$. Hence $\sum_{n=0}^M Y_n$ has an approximate normal distribution with mean $M\mu$ and variance $M\sigma^2$. The normalized random variable $\sqrt{M} \frac{\frac{1}{M} \sum_{n=0}^{M-1} Y_n - \mu}{\sigma}$ is then approximately standard normal and converges to $N(0, 1)$ when $M \rightarrow \infty$. This also holds when the *standard deviation* σ is replaced by its point estimator S_M . This point estimator is defined as $\sqrt{\frac{1}{M} \sum_{n=0}^{M-1} (Y_n - \frac{1}{M} \sum_{m=0}^{M-1} Y_m)^2}$ and converges almost surely to σ .

¹⁰When we assume $\mu \neq 0$.

¹¹Since $\lim_{M \rightarrow \infty} \frac{1}{M} \sum_{n=0}^{M-1} y_n = \mu$ (almost surely).

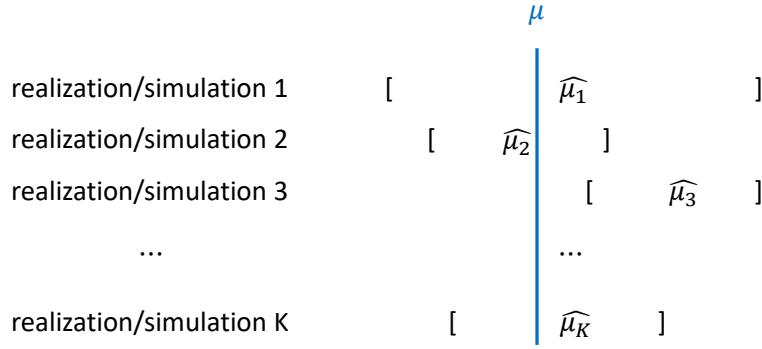


Figure B.13: Approximately fraction γ of the confidence intervals will contain μ

We now choose a so-called *confidence level* γ ($0 \leq \gamma \leq 1$) and determine value c such that $P(-c \leq N(0, 1) \leq c) = \gamma$. This is shown in Figure B.12. The bell-shaped curve, which is symmetric around 0, is the probability density function of $N(0, 1)$. The surface of the shaded area is the probability that $N(0, 1)$ takes a value between $-c$ and c and is equal to the confidence level γ . If F_{normal}^{-1} denotes the inverse of the distribution function of the standard normal distribution, then c is computed by $F_{\text{normal}}^{-1}(\frac{1+\gamma}{2})$.

Since variable $\sqrt{M} \frac{\frac{1}{M} \sum_{n=0}^{M-1} Y_i - \mu}{S_M}$ is approximately standard normal, we thus have that

$$P(-c \leq \sqrt{M} \frac{\frac{1}{M} \sum_{n=0}^{M-1} Y_i - \mu}{S_M} \leq c) \approx \gamma \quad (\text{B.61})$$

From (B.61) we can derive an important result:

$$P(\mu \in [\frac{1}{M} \sum_{n=0}^{M-1} Y_n - \frac{cS_M}{\sqrt{M}}, \frac{1}{M} \sum_{n=0}^{M-1} Y_n + \frac{cS_M}{\sqrt{M}}]) \approx \gamma \quad (\text{B.62})$$

Hence the probability that the value μ lies in the stochastic interval $[\frac{1}{M} \sum_{n=0}^{M-1} Y_n - \frac{cS_M}{\sqrt{M}}, \frac{1}{M} \sum_{n=0}^{M-1} Y_n + \frac{cS_M}{\sqrt{M}}]$ is approximately equal to the confidence level γ . Therefore this interval is called an *interval estimator* of μ .

So what can we learn from this stochastic interval? It tells us that about fraction γ of a large set of realizations of the stochastic interval will contain value μ . This is illustrated in Figure B.13. A realization of the stochastic interval is obtained by replacing the random variables Y_n with a realization y_n . We then obtain a so-called *confidence interval* which is of the following shape:

$$[\frac{1}{M} \sum_{n=0}^{M-1} y_n - \frac{cS_M}{\sqrt{M}}, \frac{1}{M} \sum_{n=0}^{M-1} y_n + \frac{cS_M}{\sqrt{M}}] \quad (\text{B.63})$$

Notice that the point in the middle of the confidence interval is formed by the point estimation $\frac{1}{M} \sum_{n=0}^{M-1} y_n (= \hat{\mu})$ of μ . The width of the interval is determined by the chosen confidence level (reflected by c), the estimation s_M of the σ , and the reciprocal of the square root of M . Notice that this interval does not necessarily contain μ , but it is said that the level of confidence that it contains μ equals γ . Typically in practice confidence levels of 95% or 99% are chosen.

Now that we understand confidence intervals, we can return to the estimation errors (B.59). Once we have obtained a confidence interval, we *assume* that it contains the true value μ and use $\hat{\mu}$ (in the middle of the confidence interval) to estimate it. Based on this we can derive a *bound* on the absolute estimation error:

$$|\mu - \hat{\mu}| \leq \frac{cS_M}{\sqrt{M}} \quad (\text{B.64})$$

Now for $M \rightarrow \infty$, cS_M converges to a constant. The error is thus inversely proportional to \sqrt{M} and converges to 0, be it slowly. So the longer we choose the simulation sequence y_0, y_1, \dots, y_M , the smaller the absolute error will be. This result gives us an answer to the question we posed earlier: how long should the simulation sequence be to obtain acceptable errors? If we target an absolute estimation error of ϵ , we simply keep on increasing the simulation sequence until $\frac{cS_M}{\sqrt{M}} \leq \epsilon$. Typical error bounds that are chosen in practice are 1% or 5%.

With respect the relative estimation errors a similar line of reasoning holds. In case the confidence interval only contains non-negative values, the error bound is as follows:

$$\frac{|\mu - \hat{\mu}|}{|\mu|} \leq \frac{cS_M/\sqrt{M}}{\hat{\mu} - cS_M/\sqrt{M}} \quad (\text{B.65})$$

Example B.20 (Throughput of an Ethernetwork). Consider a slotted Ethernetwork that behaves as a sequence of independent random variables Y_0, Y_1, \dots with a Bernoulli distribution. For each n , $P(Y_n = 1) = \mu$ and $P(Y_n = 0) = 1 - \mu$. We want to estimate parameter μ by simulation with $\gamma = 0.95$. $E(Y_n) = 1 \cdot P(Y_n = 1) + 0 \cdot P(Y_n = 0) = \mu$, so indeed the goal is to estimate $E(Y_n)$. In a simulation the following 10 realizations are obtained: 1, 1, 0, 1, 0, 0, 1, 1, 0, 0. The value μ is estimated by the average of the observations which is $\hat{\mu} = 0.5$. To determine a confidence interval we also need the standard deviation which is $s_{10} = 0.5$. In addition we require the value of c that corresponds to confidence level γ . $c = F_{\text{normal}}^{-1}(\frac{1+\gamma}{2}) \approx 1.96$, where F_{normal}^{-1} denotes the inverse of the distribution function of the standard normal distribution. The confidence interval $[\frac{1}{M} \sum_{n=0}^{M-1} y_n - \frac{cS_M}{\sqrt{M}}, \frac{1}{M} \sum_{n=0}^{M-1} y_n + \frac{cS_M}{\sqrt{M}}]$ is then $[0.19, 0.81]$ giving an absolute error bound of 0.31 and a relative error bound of 1.6. Such large error bounds are typically not acceptable in practice, and longer simulations are required to improve the accuracy.

Exercise B.34 (Throughput of an Ethernetwork - confidence levels versus error bounds). Consider the Ethernet simulation of Example B.20. One can trade-off a smaller confidence level for a decrease of the estimation error bounds.

- (a) Compute the confidence interval and the error bounds when a confidence level of 0.10 is taken.
- (b) What is the disadvantage of taking a confidence level of 0.10?

See answer.

Exercise B.35 (Throughput of an Ethernetwork - required length of simulation sequence). Consider the Ethernet simulation of Example B.20 again. Assume the true value of μ is known to be 0.6. How long should the simulation sequence be to obtain an absolute estimation error which is at most 0.001 with a confidence level of 95%?

See answer.

Exercise B.36 (Interval estimator of expected value). Derive the result of the interval estimator of μ in Equation B.62.

See answer.

Exercise B.37 (Confidence interval interpretation). A simulation to estimate some expected value μ delivers a 95% confidence interval [1.31, 4.25]. A simulation expert makes the following claim: 'The probability that μ is in the interval [1.31, 4.25] is approximately 95%. Is this claim correct?

See answer.

Exercise B.38 (Standard deviation - point estimator). Give a plausibility argument that S_M converges almost surely to σ ($= \sqrt{Var(Y_n)}$).

See answer.

B.7.2 Estimating transient properties

Now that we understood the basic theory of estimation, we can study how to apply it to Markov chains. In this subsection we will consider transient properties and defer the study of long-run properties to the next subsection. The key is to find the appropriate sequence Y_0, Y_1, \dots of independent identically distributed variables such that $E(Y_n)$ equals the property of interest. In addition it should be defined how the observations y_n are obtained based on paths generated through the Markov chain. Once appropriate variables and observations are defined, the point and interval estimations and the estimation errors follow readily from the basic estimation theory.

Example B.21 (Estimation expected rewards). Assume we like to estimate the expected reward $E(r(X_K))$ at some time K . To apply the theory in the previous subsection, we will define a sequence of independent identically distributed variables Y_0, Y_1, \dots , where each Y_i is an *independent copy* of variable $r(X_K)$. Then $E(r(X_K))$ is estimated by $\frac{1}{M} \sum_{n=0}^{M-1} y_n$ with confidence interval $[\frac{1}{M} \sum_{n=0}^{M-1} y_n - \frac{cS_M}{\sqrt{M}}, \frac{1}{M} \sum_{n=0}^{M-1} y_n + \frac{cS_M}{\sqrt{M}}]$ and corresponding bounds on absolute and relative errors. Realization y_n in the sequence y_0, y_1, \dots, y_{M-1} is obtained by first (re-)starting the Markov chain by picking randomly an initial state i , based on the initial distribution (using a (pseudo)random number generator). Then a path of length K starting in state i is generated, where the state transitions are randomly chosen based on transition probabilities (also using a (pseudo)random number generator). If this path ends in state j , $y_i = r(j)$.

Exercise B.39 (Estimation transient distributions). Explain how to estimate $\pi_i^{(K)}$ for some fixed $K \geq 0$ and state $i \in \mathcal{S}$.

See answer.

Exercise B.40 (Gambler's ruin - expected reward estimation). Consider the example of the gambler's ruin depicted in Figure B.3, where $\pi^{(0)} = [0, \frac{1}{2}, \frac{1}{2}, 0]$. We like to estimate $E(r(X_2))$ by simulation with an absolute estimation error bound of 2.5 and a confidence level of 95%.

- (a) Estimate the number of realizations (i.e. paths) required to obtain this absolute error bound. To this end you can use the CMWB to compute $\pi^{(2)}$ and $E(r(X_2))$.
- (b) Estimate $E(r(X_2))$ by simulation using the CMWB. How do you explain the difference in the obtained absolute error bound compared to (a)?

See answer.

Exercise B.41 (Gambler's ruin – converge rate central limit theorem). As a follow-up to Exercise B.40 we now like to estimate $\pi^{(10)}$ by simulation with a confidence level of 95%.

- (a) Compute $\pi^{(10)}$ using the CMWB.
- (b) Estimate $\pi^{(10)}$ by simulating 10,000 paths using the CMWB. Estimate the maximal absolute estimation error of the individual elements of the resulting vector, without using simulation.
- (c) Estimate $\pi_2^{(10)}$ by estimating $E(r(X_{10}))$ for $r = [0, 1, 0, 0]$ using simulation with an absolute error bound of 0.0001.

See answer.

Example B.22 (Estimation hitting probabilities). Assume we like to estimate hitting probability f_{ij} for some fixed states $i, j \in \mathcal{S}$. We have to define a sequence of independent identically distributed variables Y_0, Y_1, \dots such that $E(Y_n) = f_{ij}$. When we define random variable Y , such that Y takes value 1 if state j is hit from i in one or more steps and 0 otherwise, then $E(Y) = f_{ij}$. Now let each Y_n be an independent copy of Y . A realization y_n is obtained by (re-)starting the Markov chain in state i , after which a path (respecting the transition probabilities) is generated. If the path hits state j , y_n is assigned the value 1. If j cannot be reached from the current state in the path, the simulation would not terminate searching for j . Therefore we assume a (large) number K and assign y_n the value 0 if the length of the generated path exceeds K .

Exercise B.42 (Estimation hitting probability - impact of maximal path length). Consider the transition diagram depicted in Figure B.5. We like to estimate $f_{13} = \frac{1}{2}$ by simulation, yet with a maximum path length of 3.

- (a) What value will the point estimator converge to when the maximum path length is 3?
- (b) Estimate the number of realizations required to obtain a relative error bound of 0.01 for a 90% confidence level. Verify your answer using the CMWB.

See answer.

Exercise B.43 (Estimation expected cumulative reward until hit). Think of a way to estimate $\frac{f_{ij}}{f_{ij}}$ for some fixed states $i, j \in \mathcal{S}$.

See answer.

Exercise B.44 (Rover in a maze - estimation escape probability). Recall Exercise B.27 of the autonomous rover. Determine the probability that the rover finds its way out of the maze, by simulation with the CMWB only. In particular pretend that the analytical answer is unknown and do not use analytical results to estimate the number of required realizations (as we did in previous exercises).

See answer.

B.7.3 Estimating long-run properties

The estimation of long-run properties is hard for Markov chains in general. In this section we will discuss the special case for unichains and focus on the long-run expected average reward $\pi^{(\infty)} \cdot r^T$ which we will call μ .

Any unichain has a recurrent state i_r that will be visited infinitely often with probability 1. From any initial state, a unichain will always end up in the single recurrent class of which i_r

must be an element. Once arrived in any state of this class, i_r will be hit with probability one (as we proved in Exercise B.26) and will therefore keep on cycling through this state. This insight yields the basis to define a sequence of independent identically-distributed variables.

We first define random variables L_n as the length of the n^{th} cycle through i_r (where we start counting with cycle 0). The expected length $E(L_n)$ of cycle n equals the expected return time to state i_r which equals $1/\pi_{i_r}^{(\infty)}$ as we have seen in Exercise B.31. Further the (stochastic) lengths of the different cycles do not depend on each other and have the same distribution. We can therefore apply the strong law of large numbers (B.58) and conclude that $\frac{1}{M} \sum_{n=0}^{M-1} L_n$ converges almost surely to $1/\pi_{i_r}^{(\infty)}$ and is therefore a point estimator of $1/\pi_{i_r}^{(\infty)}$.

As a second step we define random variables R_n to denote the cumulative reward obtained in cycle n (where the reward in state i_r is only counted once). Also these variables are independent and identically-distributed. Notice that $E(R_n)$ is the expected cumulative reward gained before the chain hits state i_r , when starting in state i_r . Hence $E(R_n) = f_{i_r i_r}^r$. It can be shown that $f_{i_r i_r}^r = \mu/\pi_{i_r}^{(\infty)}$ (see also Exercise B.47). Therefore $\frac{1}{M} \sum_{n=0}^{M-1} R_n$ converges almost surely to $\mu/\pi_{i_r}^{(\infty)}$ and is an estimator thereof.

As a result from the previous two steps we thus obtain:

$$\frac{\sum_{n=0}^{M-1} R_n}{\sum_{n=0}^{M-1} L_n} \text{ converges almost surely to } \mu \quad (\text{B.66})$$

Hence $\frac{\sum_{n=0}^{M-1} R_n}{\sum_{n=0}^{M-1} L_n}$ is a point estimator of μ . If we replace the random variables with their realization we obtain point estimation $\frac{\sum_{n=0}^{M-1} r_n}{\sum_{n=0}^{M-1} l_n}$ which we will denote by $\hat{\mu}$.

To derive an interval estimator for μ we define the sequence Y_0, Y_1, \dots of independent identically-distributed random variables as $Y_n = R_n - \mu L_n$. For this sequence we can apply the central limit theorem to derive that

$$P(-c \leq \frac{\sum_{n=0}^{M-1} Y_n}{\sqrt{MS_M}} \leq c) \approx \gamma \quad (\text{B.67})$$

where $S_M = \sqrt{\frac{1}{M} \sum_{n=0}^{M-1} (R_n - \frac{\sum_{m=0}^{M-1} R_m}{\sum_{m=0}^{M-1} L_m} L_n)^2}$ is a point estimator of $\sigma = \sqrt{\text{Var}(Y_n)}$ and where $c = F_{\text{normal}}^{-1}(\frac{1+\gamma}{2})$. Notice that $E(Y_n) = E(R_n) - \mu E(L_n) = 0$.

Based on (B.67) we can derive the following interval estimator for μ :

$$\left[\frac{\sum_{n=0}^{M-1} R_n}{\sum_{n=0}^{M-1} L_n} - \frac{cS_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} L_n}, \frac{\sum_{n=0}^{M-1} R_n}{\sum_{n=0}^{M-1} L_n} + \frac{cS_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} L_n} \right] \quad (\text{B.68})$$

Hence a confidence interval for μ with confidence level γ has the following shape:

$$\left[\hat{\mu} - \frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n}, \hat{\mu} + \frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n} \right] \quad (\text{B.69})$$

Based on the point estimation of μ and the confidence interval, one can derive absolute and relative estimation errors. The computation of these point and interval estimations are based on realizations l_n and r_n . These are obtained by generating a single long path through the Markov chain. The initial segment of the path is ignored, i.e. from the (chosen) start state until state i_r is hit for the first time. r_n , l_n and y_n are based on the segment of the path between the n plus first and n plus second hit of i_r . The simulation can be terminated once acceptable estimation errors are obtained.

The simulation technique explained thus far, assumes that we can actually choose an explicit recurrent state and determine its visiting times during simulation. In performance modeling practice, the state space is usually not known and therefore it is not possible to choose such a recurrent state. The states can further have very complex structures, making it hard or impossible to determine whether a certain state is visited (again). In such cases, the start of the next segment or 'recurrency cycle' in the simulation path has to be determined in another way. One technique is to start a next segment when a certain reward assumes a certain value (e.g. indicating that a queue has run empty). A more popular technique, called *batching*, is to start the next segment once the number of states in the current segment reaches a predefined threshold.

Exercise B.45 (Absolute error bound long-run expected average reward).

Consider the simulation technique to estimate the long-run expected average reward.

- (a) Derive a bound of the absolute estimation error.
- (b) Show that this bound converges to 0.

See answer.

Exercise B.46 (Long-run expected average reward - confidence interval).

Consider a unichain with a reward function. A simulation delivers the following sequence of rewards: 1, 1, 2, 3, 3, 2, 4, 2, 4, 2, 3, 3, 3, 2, 3. The state in which reward 2 is obtained is a recurrent state. Compute a 95% confidence interval.

See answer.

Exercise B.47 (Expected reward until return versus long-run expected average reward). One can show that for any recurrent state i , $f_{ii}^r = \pi^{(\infty)} \cdot r^T / \pi_i^{(\infty)}$. This property was used as a basis to establish Equation B.66. Show that this property

holds for state 2 of the Markov chain with transition probability matrix

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{5}{6} & 0 & \frac{1}{6} \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

and reward function r defined by $r(1) = 3$, $r(2) = 5$, $r(3) = 8$ and $r(4) = 2$.
See answer.

Exercise B.48 (Interval estimator long-run expected average reward). Derive the interval estimator for $\pi^{(\infty)} \cdot r^T$ given in Equation B.68.

See answer.

Exercise B.49 (Estimation long-run expected fraction of time spent in a state). Let P^∞ be the Cezàro limiting matrix of a unichain. Explain how to estimate P_{ij}^∞ for fixed states $i, j \in \mathcal{S}$.

See answer.

Exercise B.50 (Estimation Cezàro limiting distribution of a non-unichain).

Consider Example B.18 in which the Cezàro limiting distribution is computed corresponding to the transition diagram in Figure B.5 for $\pi^{(0)} = [\frac{1}{4}, \frac{3}{4}, 0, 0, 0, 0, 0, 0]$. This limit is given by $\pi^{(0)} P^\infty = [0, 0, 0, \frac{21}{80}, \frac{21}{80}, \frac{19}{90}, \frac{19}{180}, \frac{19}{120}]$. Make a couple of attempts to estimate this limit using CMWB. Explain your findings.

See answer.

Exercise B.51 (Video application - long-run average buffer occupancy). Consider the Markov chain of the video application in Exercise B.24 for $p = \frac{1}{2}$. Estimate by means of CMWB the expected long-run buffer occupancy level. Use a 95% confidence level and a relative error bound of 0.01.

See answer.

B.8 Answers to Exercises

Exercise B.1 (Wealthy gambler - expected reward).

- (a) The game can be modelled as a stochastic process X_0, X_1, \dots with state-space $\mathcal{S} = \{0, 1, \dots, 36\}$, where state $i \in \mathcal{S}$ corresponds to pocket number i . X_n represents the pocket that the roulette ball enters after spin $n + 1$. The probability that the roulette ball ends in pocket i equals $\frac{1}{37}$. Hence $P(X_n = i) = \pi_i^{(n)} = \frac{1}{37}$.
- (b) The players earns €500 when the ball enters one of the pockets 1, 3, …, 35, and loses €500 otherwise. This can be modelled by a reward $r : \mathcal{S} \rightarrow \mathbb{R}$ with $r(i) = 500$ for $i = 1, 3, \dots, 35$ and $r(i) = -500$ for $i = 0, 2, 4, \dots, 36$.
- (c) The expected reward at any time n (so also for $n = 4$) is given by $E(r(X_n)) = \pi^{(n)} r^T = 18 \cdot \frac{1}{37} \cdot 500 + 19 \cdot \frac{1}{37} \cdot -500 = -\frac{500}{37} \approx -13.5 \text{ €}$.
- (d) The loss probability per spin equals $\frac{19}{37}$. Since random variables X_n are independent, the probability of losing 20 times in a row is given by $\frac{19}{37}^{20} \approx 1.6 \cdot 10^{-6}$.

Exercise B.2 (Time-slotted Ethernetwork - throughput).

- (a) The communication system can be modelled as a stochastic process X_0, X_1, \dots with state-space $\mathcal{S} = \{0, 1, 2\}$, where state $i \in \mathcal{S}$ corresponds to the number of packets offered to the Ethernetwork. The probability distribution is as follows: $P(X_n = 0) = \frac{2}{3} \cdot \frac{4}{5} = \frac{8}{15}$, $P(X_n = 1) = \frac{1}{3} \cdot \frac{4}{5} + \frac{2}{3} \cdot \frac{1}{5} = \frac{6}{15}$ and $P(X_n = 2) = \frac{1}{3} \cdot \frac{1}{5} = \frac{1}{15}$. Hence $\pi^{(n)} = [\frac{8}{15}, \frac{6}{15}, \frac{1}{15}]$.
- (b) The number of transmitted frames during a time slot can be modelled by reward $r : \mathcal{S} \rightarrow \mathbb{R}$ defined by $r(0) = r(2) = 0$ and $r(1) = 1$.
- (c) The probability that the medium does not transmit during time slot n is given by $P(r(X_n) = 0)$. This expression is computed by $\sum_{i \in \mathcal{S}, r(i)=0} P(X_n = i) = P(X_n = 0) + P(X_n = 2) = \frac{8}{15} + \frac{1}{15} = \frac{3}{5}$.
- (d) The expected number of transmitted frames per time slot is computed by $E(r(X_n)) = \pi^{(n)} r^T = [\frac{8}{15}, \frac{6}{15}, \frac{1}{15}] [0, 1, 0]^T = \frac{6}{15} = \frac{2}{5}$ frames per time slot.

Exercise B.3 (Expected reward - computation). The expected reward at time n is given by $E(r(X_n)) = \sum_{v \in \mathcal{R}} v \cdot P(r(X_n) = v)$. By using (B.4) this expression is rewritten as $\sum_{v \in \mathcal{R}} v \cdot \sum_{i \in \mathcal{S}, r(i)=v} P(X_n = i)$. By using (B.2) we obtain $\sum_{v \in \mathcal{R}} v \cdot \sum_{i \in \mathcal{S}, r(i)=v} \pi_i^{(n)} = \sum_{v \in \mathcal{R}} \sum_{i \in \mathcal{S}, r(i)=v} v \cdot \pi_i^{(n)} = \sum_{v \in \mathcal{R}} \sum_{i \in \mathcal{S}, r(i)=v} r(i) \cdot \pi_i^{(n)} = \sum_{i \in \mathcal{S}} r(i) \cdot \pi_i^{(n)}$. The latter expression equals $\pi^{(n)} r^T$ when r is considered to be the row vector $[r(1), r(2), \dots, r(N)]$.

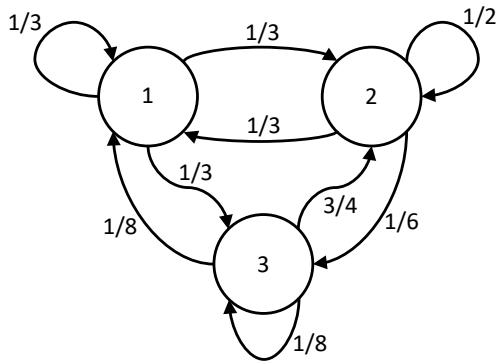


Figure B.14: Transition diagram of three-state Markov chain

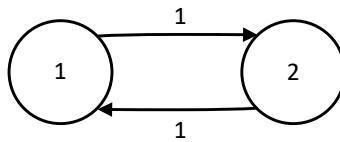


Figure B.15: Transition diagram of two-state Markov chain

Exercise B.4 (Transition diagram to matrix). The transition probability matrix is given by

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & 0 & 0 \end{bmatrix}$$

Exercise B.5 (Matrix to transition diagram). The transition diagram is depicted in Figure B.14.

Exercise B.6 (Markov chains - dependent and non-identically distributed variables).

- (a) When the chain is in state 1, it will transition to state 2 with probability 1. Vice versa, when the chain is in state 2, it will transition to state 1 with probability 1. Hence the transition probability matrix is given $P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. The corresponding transition diagram is shown in Figure B.15.
- (b) At time 0, the chain visits state 0 with probability 1. Therefore $\pi^{(0)} = [1, 0]$.
- (c) After visiting state 0, the chain jumps to state 1 with probability 1. At time 1, the chain thus visits state 1 with probability 1 and therefore $\pi^{(1)} = [0, 1]$.
- (d) Distributions $\pi^{(0)}$ and $\pi^{(1)}$ are different and therefore X_0 and X_1 are not identically distributed.

- (e) Assume X_1 and X_0 are independent. Then for all $i, j \in \{1, 2\}$, $P(X_1 = j | X_0 = i) = P(X_1 = j)$. In particular, for $i = 2$ and $j = 2$, we then have $P(X_1 = 2 | X_0 = 2) = P(X_1 = 2)$. But $P(X_1 = 2 | X_0 = 2) = P_{22} = 0$ and $P(X_1 = 2) = \pi_2^{(1)} = 1$, so we have a contradiction. Therefore X_1 and X_0 are dependent variables.

Exercise B.7 (Gambler's ruin - probability distributions).

- (a) The chain starts in state 2 (with probability 1). After one transition, it will be in state 1 with probability $\frac{1}{2}$ and in state 3 with probability $\frac{1}{2}$. Therefore $\pi^{(1)} = [\frac{1}{2}, 0, \frac{1}{2}, 0]$.
- (b) The probability that the chain is in state 1 at time 2 equals the probability that the chain is in state 1 at time 1 times 1 plus the probability that the chain is in state 2 at time 1 times $\frac{1}{2}$. Hence $\pi_1^{(2)} = \pi_1^{(1)} \cdot 1 + \pi_2^{(1)} \cdot \frac{1}{2} = \frac{1}{2}$. With a similar line of thought we obtain $\pi_2^{(2)} = \frac{1}{4}$, $\pi_3^{(2)} = 0$ and $\pi_4^{(2)} = \frac{1}{4}$. Hence $\pi^{(2)} = [\frac{1}{2}, \frac{1}{4}, 0, \frac{1}{4}]$.

Exercise B.8 (Markov chains - independent identically distributed variables).

- (a) Using a similar line of thought as used in Exercise B.7, we find that $\pi^{(n)} = [\frac{1}{2}, \frac{1}{2}]$ for all $n = 0, 1, \dots$. Therefore X_n and X_{n+1} are identically distributed for all $n = 0, 1, \dots$.
- (b) We have to show that $P(X_{n+1} = j | X_n = i) = P(X_{n+1} = j)$ for all $i, j \in \{1, 2\}$. Now $P(X_{n+1} = j | X_n = i) = P_{ij} = \frac{1}{2}$ and $P(X_{n+1} = j) = \pi_j^{(n+1)} = \frac{1}{2}$ (for all $i, j \in \{1, 2\}$) from which the result follows.

Exercise B.9 (Probability distributions via matrix algebra).

- (a) $P(X_1 = 1) = \pi_1^{(1)} = (\pi^{(0)}P^1)_1 = ([1, 0] \cdot \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ 1 & 0 \end{bmatrix})_1 = [\frac{1}{3}, \frac{2}{3}]_1 = \frac{1}{3}$.
- (b) $P(X_2 = 1) = \pi_1^{(2)} = (\pi^{(0)}P^2)_1 = ([1, 0] \cdot \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ 1 & 0 \end{bmatrix}^2)_1 = ([1, 0] \cdot \begin{bmatrix} \frac{7}{9} & \frac{2}{9} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix})_1 = [\frac{7}{9}, \frac{2}{9}]_1 = \frac{7}{9}$.
- (c) $P(X_{20} = 1) = \pi_1^{(20)} = (\pi^{(0)}P^{20})_1 = ([1, 0] \cdot \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ 1 & 0 \end{bmatrix}^{20})_1 \approx ([1, 0] \cdot \begin{bmatrix} 0.60 & 0.40 \\ 0.60 & 0.40 \end{bmatrix})_1 = [0.60, 0.40]_1 = 0.60$.
- (d) For very large n , $\begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ 1 & 0 \end{bmatrix}^n \approx \begin{bmatrix} 0.60 & 0.40 \\ 0.60 & 0.40 \end{bmatrix}$. Hence the probability that the Markov chain is in state 1 after a very large number of transitions is (approximately) 0.60.

Exercise B.10 (N-step transition probabilities).

(a) The probability is given by $P_{12}^5 = \begin{bmatrix} \frac{1281}{3125} & \frac{1844}{3125} \\ \frac{461}{625} & \frac{164}{625} \end{bmatrix}_{12} = \frac{1844}{3125} \approx \begin{bmatrix} 0.41 & 0.59 \\ 0.74 & 0.26 \end{bmatrix}_{12} = 0.59.$

- (b) The probability is computed by summing the probabilities of all paths of from state 1 to state 2 of length 5. There exist five such paths:

- path 1, 1, 1, 1, 1, 2 with probability $\frac{1}{5}^4 \cdot \frac{4}{5}$
- path 1, 1, 1, 2, 1, 2 with probability $\frac{1}{5}^2 \cdot \frac{4}{5}^2$
- path 1, 1, 2, 1, 1, 2 with probability $\frac{1}{5}^2 \cdot \frac{4}{5}^2$
- path 1, 2, 1, 1, 1, 2 with probability $\frac{1}{5}^2 \cdot \frac{4}{5}^2$
- path 1, 2, 1, 2, 1, 2 with probability $\frac{4}{5}^3$

Adding the probabilities yields $\frac{1844}{3125} \approx 0.59$, so as expected we obtain the same answer as in (a).

Exercise B.11 (Gambler's ruin - n-step probabilities and expected reward).

- (a) With an initial capital of €200, the Markov chain is in state 3. The gambler is broke in state 1. Therefore the answer is $P_{31}^{100} \approx 0.333$.
- (b) Since there exist no paths of length 1000 from state 2 to state 3, this probability is 0.
- (c) The expected amount of cash is given by (see equation (B.5)) $\pi^{(16)} r^T = \{ \text{ see equation (B.15)} \} \pi^{(0)} P^{16} r^T \approx €150$.

Exercise B.12 (Gambler's ruin - dependent and non-identically distributed variables).

- (a) $\pi^{(3)} = \pi^{(0)} P^3 = [\frac{15}{32}, \frac{1}{32}, \frac{1}{32}, \frac{15}{32}] \approx [0.46875, 0.03125, 0.03125, 0.46875]$ and $\pi^{(6)} = \pi^{(0)} P^6 = [\frac{127}{256}, \frac{1}{256}, \frac{1}{256}, \frac{127}{256}] \approx [0.49609375, 0.00390625, 0.00390625, 0.49609375]$. These distributions are different and hence X_3 and X_6 are not identically distributed.

- (b) We have to show (see equation (B.7)) that $P(X_6 = j \mid X_3 = i) \neq P(X_6 = j)$ for some $i, j \in \{1, 2, 3, 4\}$. Now $P(X_6 = j \mid X_3 = i) = \{ \text{ see equation (B.16)} \} P_{ij}^3$
- $$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{5}{8} & 0 & \frac{1}{8} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{8} & 0 & \frac{5}{8} \\ 0 & 0 & 0 & 1 \end{bmatrix}_{ij} \approx \begin{bmatrix} 1.000 & 0.000 & 0.000 & 0.000 \\ 0.625 & 0.000 & 0.125 & 0.250 \\ 0.250 & 0.125 & 0.000 & 0.625 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix}_{ij} \text{ and } P(X_6 = j) = \pi_j^{(6)}. \text{ Clearly } P_{ij}^3 \neq \pi_j^{(6)} \text{ even for all } i, j \in \{1, 2, 3, 4\} \text{ and hence } X_3 \text{ and } X_6 \text{ are dependent.}$$

Exercise B.13 (Queue in time-slotted communication network - modeling and transient analysis).

- (a) We can model the queue by a Markov chain X_0, X_1, \dots with state-space $\mathcal{S} = \{1, 2, 3, 4\}$ and reward $r : \mathcal{S} \rightarrow \mathbb{R}$ such that $r(i) = i - 1$, where $r(X_i)$ denotes the number of packets present in the queue at the beginning of time slot i . The probability matrix is given

$$\text{by } P = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{6} & \frac{1}{12} & \frac{1}{7} & 0 \\ 0 & \frac{1}{6} & \frac{1}{12} & \frac{1}{4} \\ 0 & 0 & \frac{1}{4} & \frac{3}{4} \end{bmatrix}, \text{ and the initial distribution by } \pi^{(0)} = [1, 0, 0, 0]. \text{ Here } P_{23}, \text{ for}$$

instance, is the probability that the occupancy of the queue is increased from $r(2) = 1$ to $r(3) = 2$. This probability equals the probability that one packet is offered and no packet is retrieved, which is $\frac{1}{3} \cdot \frac{3}{4} = \frac{1}{4}$.

- (b) The probability that the queue is full at the beginning of time slot 15 is given by $\pi_4^{(15)} = \pi^{(0)} P_4^{15} \approx 0.29162365$.
- (c) The answer equals P_{31}^{20} , i.e. the probability that the chain transits from state 3 to state 1 in 20 steps. $P_{31}^{20} \approx 0.109$.
- (d) The expected occupancy is given by $\pi^{(5)} r^T = \pi^{(0)} P^5 r^T \approx 1.1189$.

Exercise B.14 (Independent identically distributed variables as Markov chain).

- (a) Assume that $\pi^{(0)} = [p_1, p_2]$ where $p_1, p_2 \in [0, 1]$ such that $p_1 + p_2 = 1$. Define probability matrix $P = \begin{bmatrix} p_1 & p_2 \\ p_1 & p_2 \end{bmatrix}$.
- (b) First note that $P^n = P$ for all $n = 1, 2, \dots$. From this it follows that $\pi^{(n)} = \pi^{(0)} P^n = [p_1, p_2]P = [p_1, p_2]$ for all $n = 0, 1, \dots$. Hence X_n and X_m are identically distributed for all $n, m \geq 0$. We still have to show that X_n and X_m are independent when $m \neq n$. For this assume $n = m + k$ for some $k > 0$. We have to show that $P(X_{m+k} = j | X_m = i) = P(X_{m+k} = j)$ for all $i, j \in \{1, 2\}$. Now $P(X_{m+k} = j | X_m = i) = \{\text{by using (B.16)\}}$ $P_{ij}^k = P_{ij} = \pi_j^{(m+k)} = P(X_{m+k} = j)$ which completes the proof.

Exercise B.15 (Classes of a Markov chain). The classes are found by considering the individual states. State 2 is accessible from state 1 but not the other way around. Therefore state 1 only communicates with itself and forms class $\{1\}$. State 2 communicates with states 3 and 4, and therefore these states together form class $\{2, 3, 4\}$. State 5 is isolated; no other states are accessible from state 5 and state 5 cannot be accessed by any other state. Therefore state 5 only communicates with itself and forms class $\{5\}$. Together these classes form a partition of the state space. This partition is $\{\{1\}, \{2, 3, 4\}, \{5\}\}$.

Exercise B.16 (State accessibility versus paths).

- ' \Rightarrow ' Assume $i \rightarrow j$. Then by definition $P_{ij}^n > 0$ for some $n \geq 0$. By equation (B.19) we then have that the sum of the probabilities of all paths of length n from i to j is at least 0. Therefore there exists at least one path of length n from i to j .
- ' \Leftarrow ' Assume a path exists between i and j . If this path has length n , the probability of all paths of length n from i to j is at least 0. By equation (B.19) we then must have that $P_{ij}^n > 0$ and therefore $i \rightarrow j$.

Exercise B.17 (Communicating states - equivalence relation). We have to show that \leftrightarrow is reflexive, symmetric and transitive:

- Since $i \rightarrow i$, we have by definition $i \leftrightarrow i$. Hence \leftrightarrow is reflexive.
- Assume $i \leftrightarrow j$. By definition we then have $i \rightarrow j$ and $j \rightarrow i$ and thus also $j \leftrightarrow i$. Hence \leftrightarrow is symmetric.
- Assume $i \leftrightarrow j$ and $j \leftrightarrow k$. Then $i \rightarrow j$ and $j \rightarrow k$ and thus a path exists from i to j and a path exists from j to k . Concatenation yields a path from i to k and thus $i \rightarrow k$. Vice versa we also have $k \rightarrow i$ and hence $i \leftrightarrow k$. Hence \leftrightarrow is transitive.

Exercise B.18 (Recurrent versus transient classes).

Exercise B.18 (Recurrent versus transient classes). Class $\{1\}$ has an outgoing transition to class $\{2, 3, 4\}$ and is therefore transient. State 1 is therefore also transient. Class $\{2, 3, 4\}$ is closed (i.e. has no outgoing transitions) and is thus recurrent. Hence each of the states 2, 3 and 4 is recurrent. Class $\{5\}$ is also closed and is therefore recurrent. Hence state 5 is recurrent as well.

Exercise B.19 (Computing return probabilities through paths).

- A single path of length ≥ 1 exists from state 1 to state 1 which is path 1, 1. $P(1, 1) = \frac{1}{3}$ and hence $f_{11} = \frac{1}{3}$. Since $f_{11} < 1$, state 1 is transient.
- Two paths of length ≥ 1 exist from state 2 to state 2 which are paths 2, 2 and 2, 3, 4, 2. $P(2, 2) = \frac{1}{2}$ and $P(2, 3, 4, 2) = \frac{1}{2}$ and hence $f_{22} = 1$. Since $f_{22} = 1$, state 2 is recurrent.

Exercise B.20 (Periodic versus aperiodic classes).

- Class $\{1\}$ is transient; for transient classes we have not defined the concept of periodicity. Class $\{2, 3, 4, 5\}$ is recurrent. To determine the period of this class, we can pick any state and only have to consider the paths that do not visit that state in between. For state 3 only two such paths exist, namely 3, 4, 5, 3 and 3, 4, 2, 3, both with length 3. Hence the period of class $\{2, 3, 4, 5\}$ equals 3.
- The chain has a single recurrent class which is periodic. Hence this chain is a non-ergodic unichain.

Exercise B.21 (Aperiodic states are eventually visited).

- (a) This chain a single recurrent, aperiodic class. Hence this is an ergodic unichain.
- (b) For P^5, P^6, \dots this follows from equation (B.33), where the number of states (N) equals 3. It states that for $n \geq (3 - 1)^2 + 1 = 5$, $P_{ij}^n > 0$ for all $i, j \in \{1, 2, 3\}$.

For $n = 4$ it follows from explicitly computing P^4 yielding the matrix $\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{8} & \frac{1}{4} & \frac{5}{8} \\ \frac{5}{16} & \frac{1}{8} & \frac{9}{16} \end{bmatrix} \approx \begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0.125 & 0.25 & 0.625 \\ 0.312 & 0.125 & 0.562 \end{bmatrix}$ having only positive elements.

Exercise B.22 (Computing return probabilities through equations).

- (a) The equations are: $x_1 = \frac{1}{3} + \frac{2}{3}x_2$, $x_2 = \frac{1}{2}x_2 + \frac{1}{2}x_3$, $x_3 = x_4$, $x_4 = x_2$ and $x_5 = x_5$. Solving for x_1 , x_2 , x_3 and x_4 yields $x_1 = \frac{1}{3}$, $x_2 = 0$, $x_3 = 0$ and $x_4 = 0$. The equation $x_5 = x_5$ has infinitely many solutions, but the least non-negative solution is 0. Notice that we can quickly determine the values of x_2 , x_3 , x_4 and x_5 to be zero by realising that state 1 is not accessible from states 2, 3, 4 and 5 respectively. We thus have that $f_{11} = \frac{1}{3}$; the same as the answer to Exercise B.19(a).
- (b) The equations are $x_1 = \frac{1}{3}x_1 + \frac{2}{3}$, $x_2 = \frac{1}{2} + \frac{1}{2}x_3$, $x_3 = x_4$, $x_4 = x_2$ and $x_5 = x_5$. Solving for x_1 , x_2 , x_3 and x_4 yields $x_1 = 1$, $x_2 = 1$, $x_3 = 1$ and $x_4 = 1$. The least non-negative solution to $x_5 = x_5$ is 0. Hence $f_{22} = 1$, similar to the outcome of Exercise B.19(b).

Exercise B.23 (Infinite closed classes are not necessarily recurrent).

- (a) Following Equation B.36 we obtain the following equations: $x_1 = \frac{1}{2} + \frac{1}{2}x_2$, $x_2 = \frac{1}{2} + \frac{1}{2}x_3$, and $x_n = \frac{1}{2}x_{n-1} + \frac{1}{2}x_{n+1}$ (for $n \geq 3$). Notice that we obtain a solution when all variables are set to 1, in which case $x_1 = 1$. However, the hitting probability f_{11} forms the *least non-negative* solution. To show that no smaller solution than $x_1 = 1$ exists, assume that $x_1 = 1 - \epsilon$ for some $\epsilon > 0$. Then $x_2 = 1 - 2\epsilon$, $x_3 = 1 - 4\epsilon$ and in general $x_n = 1 - 2(n-1)\epsilon$ (for $n \geq 2$). But then for large enough n , x_n would be negative and thus $x_1 = 1 - \epsilon$ does not result in an overall non-negative solution. Hence for $x_1 = 1$ the least non-negative solution is obtained and thus $f_{11} = 1$.
- (b) Following Equation B.36 we obtain the following equations: $x_1 = \frac{1}{3} + \frac{2}{3}x_2$, $x_2 = \frac{1}{3} + \frac{2}{3}x_3$, and $x_n = \frac{1}{3}x_{n-1} + \frac{2}{3}x_{n+1}$ (for $n \geq 3$). It is easy to check through substitution that $x_1 = \frac{2}{3}$ and $x_n = \frac{1}{2}^n$ (for $n \geq 2$) are a solution to these equations. The hitting probability f_{11} is the least non-negative solution which is therefore at most $\frac{2}{3}$. Hence $f_{11} < 1$.

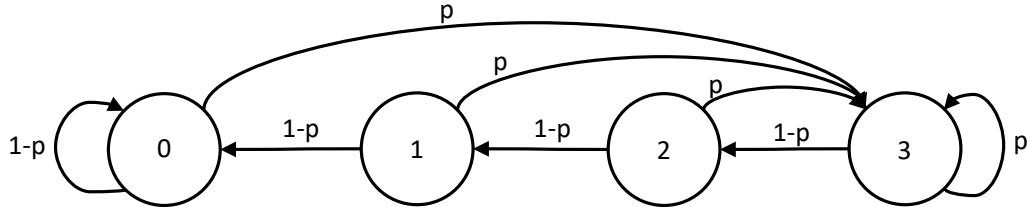


Figure B.16: Transition diagram of movie application

Exercise B.24 (Hiccups in a video application).

- (a) We can model this system as a Markov chain X_0, X_1, \dots with state-space $\{0, 1, 2, 3\}$, where X_n represents the number of frames present in the buffer just before the next clock event is raised. A transition diagram is shown in Figure B.16. From each state a transition with probability p exists to state 3, corresponding to the refill of the buffer. In case the buffer is not refilled (with probability $1-p$), the buffer occupancy decreases by 1 (except when the buffer is empty).
- (b) We have to compute the expected number of steps to transition from state 3 to state 0. We do so by assigning a reward value of 1 to each state and compute f_{30}^r/f_{30} . Now it is not hard to find out that $f_{i0} = 1$ for each i . To compute f_{30}^r we solve the following system of equations (see (B.39)): $x_0 = 1 + px_3$, $x_1 = 1 + px_3$, $x_2 = 1 + (1-p)x_1 + px_3$, $x_3 = 1 + (1-p)x_2 + px_3$. Solving yields $x_0 = \frac{1}{(1-p)^3}$, $x_1 = \frac{1}{(1-p)^3}$, $x_2 = \frac{2-p}{(1-p)^3}$ and $x_3 = \frac{3-3p+p^2}{(1-p)^3}$. Hence $f_{30}^r = \frac{3-3p+p^2}{(1-p)^3}$.
- (c) Each frame takes $\frac{1}{60}$ seconds. So the expected time until a hiccup occurs is $\frac{3-3p+p^2}{60(1-p)^3}$ seconds. For $p = 0.983239$, this expression is approximately equal to 3600. Since the expression is increasing in p , the answer is $[0.983239, 1]$.

Exercise B.25 (Hitting a state versus hitting a singleton state set). Fix $j \in \mathcal{S}$. Let x_i and y_i ($i \neq j$) denote the variables corresponding to the linear equations in (B.36) and (B.41) respectively (where $H = \{j\}$ in (B.41)). Then $x_i = P_{ij} + \sum_{k \in \mathcal{S} \setminus \{j\}} P_{ik}x_k$ and $y_i = \sum_{k \in \mathcal{S}} P_{ik}y_k$. Since $y_j = 1$ (see (B.41)), the latter equations can be rewritten to $y_i = P_{ij} + \sum_{k \in \mathcal{S} \setminus \{j\}} P_{ik}y_k$. Hence the systems of equations are the same, up to the use of variable names, containing all variables except for x_j and y_j . Hence their least solutions are the same as well.

Exercise B.26 (Hitting recurrent states with probability 1). Let C be a recurrent class and fix $j \in C$. Consider the system of equations in (B.36): $x_i = P_{ij} + \sum_{k \in \mathcal{S} \setminus \{j\}} P_{ik}x_k$. We will show that these equations have a unique solution, where $x_i = 1$ for all $i \in C$. In case $i = j$, the fact that $x_j = 1$ follows from (B.27). In that case we have $x_j = P_{jj} + \sum_{k \in \mathcal{S} \setminus \{j\}} P_{jk}x_k$. Since all successor states of j are in C we can rewrite this equation to $x_j = P_{jj} + \sum_{k \in C \setminus \{j\}} P_{jk}x_k$. Since $\sum_{k \in C} P_{jk} = 1$ and $x_j = 1$ we must have $x_k = 1$ for each state k that is accessible from j . We can perform the above steps for each of these successor

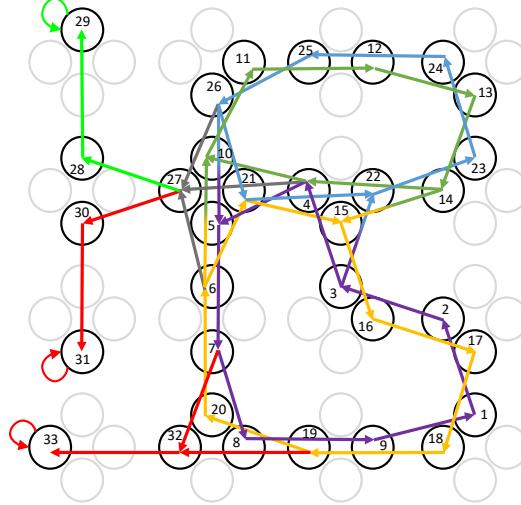


Figure B.17: Transition diagram of rover in maze problem

states (and the successors thereof, etcetera) until all states in C have been considered. Hence $x_i = 1$ (and thus also $f_{ij} = 1$) for each $i \in C$.

Exercise B.27 (Rover in a maze). We first create a Markov chain. The basic idea is to create a state for each cell of the maze. However, since the rover cannot return to the cell it just came from, in some way information about the previous cell has to be remembered. The transition diagram in Figure B.17 presents a solution which models the decision for the next cell to visit. To this end each cell corresponds to four different states. The fact that the rover starts to ride in the upward direction, corresponds to the top-most state corresponding to initial cell (i.e. the cell in which the rover starts). When the rover arrives in the cell above the initial cell, as a next step it must move to the right. Hence a transition is made from the top-most state of the initial cell to the left-most state of the cell above it. Certain states are never used. These states are given a colour grey in the figure. The other states are given a black colour and have a number between 1 and 33. Since the outgoing transition probabilities for each state are the same, these are omitted from the figure. Transitions that belong to a cyclic path in the diagram are given the same colour. The Markov chain has three recurrent classes: $\{29\}$, $\{31\}$ and $\{33\}$. All states that are not in these recurrent classes are transient. Eventually the Markov chain will visit one of three recurrent classes and stay there forever. The goal is to end up in recurrent class $\{29\}$. In that case the rover has found the exit. In case the other classes are hit, the rover is stuck in the maze.

- (a) The probability for the rover to find its way out of the maze is given by hitting probability h_{1H} , where $H = \{29\}$. The equations are given by $x_1 = x_2$, $x_2 = x_3$, $x_3 = \frac{1}{2}x_4 + \frac{1}{2}x_{22}$, $x_4 = \frac{1}{3}x_5 + \frac{1}{3}x_{27} + \frac{1}{3}x_{10}$, $x_5 = x_7$, $x_6 = \frac{1}{3}x_{10} + \frac{1}{3}x_{21} + \frac{1}{3}x_{27}$, $x_7 = \frac{1}{2}x_8 + \frac{1}{2}x_{32}$, $x_8 = x_9$, $x_9 = x_1$, $x_{10} = x_{11}$, $x_{11} = x_{12}$, $x_{12} = x_{13}$, $x_{13} = x_{14}$, $x_{14} = \frac{1}{2}x_4 + \frac{1}{2}x_{15}$, $x_{15} = x_{16}$, $x_{16} = x_{17}$, $x_{17} = x_{18}$, $x_{18} = x_{19}$, $x_{19} = \frac{1}{2}x_{20} + \frac{1}{2}x_{32}$, $x_{20} = x_6$, $x_{21} = \frac{1}{2}x_{15} + \frac{1}{2}x_{22}$, $x_{22} = x_{23}$, $x_{23} = x_{24}$, $x_{24} = x_{25}$, $x_{25} = x_{26}$, $x_{26} = \frac{1}{3}x_5 + \frac{1}{3}x_{21} + \frac{1}{3}x_{27}$, $x_{27} = \frac{1}{2}x_{28} + \frac{1}{2}x_{30}$, $x_{28} = x_{29}$, $x_{29} = 1$, $x_{30} = 0$, $x_{31} = 0$, $x_{32} = 0$ and $x_{33} = 0$. Solving yields $x_1 = \frac{11}{38}$, $x_2 = \frac{11}{38}$, $x_3 = \frac{11}{38}$, $x_4 = \frac{11}{38}$, $x_5 = \frac{11}{76}$, $x_6 = \frac{6}{19}$, $x_7 = \frac{11}{76}$, $x_8 = \frac{11}{38}$, $x_9 = \frac{11}{38}$, $x_{10} = \frac{17}{76}$,

$x_{11} = \frac{17}{76}, x_{12} = \frac{17}{76}, x_{13} = \frac{17}{76}, x_{14} = \frac{17}{76}, x_{15} = \frac{3}{19}, x_{16} = \frac{3}{19}, x_{17} = \frac{3}{19}, x_{18} = \frac{3}{19}, x_{19} = \frac{3}{19}, x_{20} = \frac{6}{19}, x_{21} = \frac{17}{76}, x_{22} = \frac{11}{38}, x_{23} = \frac{11}{38}, x_{24} = \frac{11}{38}, x_{25} = \frac{11}{38}, x_{26} = \frac{11}{38}, x_{27} = \frac{1}{2}, x_{28} = 1, x_{29} = 1, x_{30} = 0, x_{31} = 0, x_{32} = 0, x_{33} = 0$ and thus $h_{1H} = \frac{11}{38}$.

- (b) The expected number of cells the rover visits until it finds the exit cell is given by h_{1H}^r/h_{1H} , where reward r is such that it assigns the value 1 to each state. To compute h_{1H}^r we have to solve the following equations: $y_1 = r(1)x_1 + y_2, y_2 = r(2)x_2 + y_3, y_3 = r(3)x_3 + \frac{1}{2}y_4 + \frac{1}{2}y_{22}, y_4 = r(4)x_4 + \frac{1}{3}y_5 + \frac{1}{3}y_{27} + \frac{1}{3}y_{10}, y_5 = r(5)x_5 + y_7, y_6 = r(6)x_5 + \frac{1}{3}y_{10} + \frac{1}{3}y_{21} + \frac{1}{3}y_{27}, y_7 = r(7)x_7 + \frac{1}{2}y_8 + \frac{1}{2}y_{32}, y_8 = r(8)x_8 + y_9, y_9 = r(9)x_9 + y_1, y_{10} = r(10)x_{10} + y_{11}, y_{11} = r(11)x_{11} + y_{12}, y_{12} = r(12)x_{12} + y_{13}, y_{13} = r(13)x_{13} + y_{14}, y_{14} = r(14)x_{14} + \frac{1}{2}y_4 + \frac{1}{2}y_{15}, y_{15} = r(15)x_{15} + y_{16}, y_{16} = r(16)x_{16} + y_{17}, y_{17} = r(17)x_{17} + y_{18}, y_{18} = r(18)x_{18} + y_{19}, y_{19} = r(19)x_{19} + \frac{1}{2}y_{20} + \frac{1}{2}y_{32}, y_{20} = r(20)x_{20} + y_6, y_{21} = r(21)x_{21} + \frac{1}{2}y_{15} + \frac{1}{2}y_{22}, y_{22} = r(22)x_{22} + y_{23}, y_{23} = r(23)x_{23}y_{24}, y_{24} = r(24)x_{24} + y_{25}, y_{25} = r(25)x_{25} + y_{26}, y_{26} = r(26)x_{26} + \frac{1}{3}y_5 + \frac{1}{3}y_{21} + \frac{1}{3}y_{27}, y_{27} = r(27)x_{27} + \frac{1}{2}y_{28} + \frac{1}{2}y_{30}, y_{28} = r(28)x_{28} + y_{29}, y_{29} = 0, y_{30} = 0, y_{31} = 0, y_{32} = 0$ and $y_{33} = 0$. Solving yields $y_1 = \frac{1489}{361}$ and thus $h_{1H}^r = \frac{1489}{361}$. Therefore $h_{1H}^r/h_{1H} = \frac{2978}{209}$.
- (c) The expected number of times the rover visits the initial cell before it finds the exit cell is using the exact same equations as in (b). The only difference concerns the rewards assigned; in case the reward equals 1 for state 1 and 0 for all the other states. In that case solving results in $h_{1H}^r = \frac{132}{361}$ and thus $h_{1H}^r/h_{1H} = \frac{24}{19}$.

Exercise B.28 (Limiting matrix ergodic unichain).

$$(a) P^5 \approx \begin{bmatrix} 0.125 & 0.250 & 0.625 \\ 0.312 & 0.125 & 0.562 \\ 0.281 & 0.312 & 0.406 \end{bmatrix}, P^{10} \approx \begin{bmatrix} 0.270 & 0.258 & 0.473 \\ 0.236 & 0.27 & 0.494 \\ 0.247 & 0.236 & 0.517 \end{bmatrix}, P^{15} \approx \begin{bmatrix} 0.247 & 0.247 & 0.505 \\ 0.253 & 0.247 & 0.500 \\ 0.250 & 0.253 & 0.497 \end{bmatrix}$$

- (b) First notice the matrix corresponds to an ergodic unichain and therefore the limit P^∞ exists. The rows of P^∞ are equal to the unique solution of the balance equations. These equations are: $x_1 = \frac{1}{2}x_3, x_2 = x_1, x_3 = x_2 + \frac{1}{2}x_3, x_1 + x_2 + x_3 = 1$. Solving yields: $x_1 = \frac{1}{4}, x_2 = \frac{1}{4}, x_3 = \frac{1}{2}$. Hence $P^\infty = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{bmatrix}$. Notice that P^{15} is already a good approximation of P^∞ .

Exercise B.29 (Video application - limiting distribution).

- (a) Notice that the Markov chain is an ergodic unichain since $0 < p < 1$. Therefore the limiting distribution exists. We can find it by solving the balance equations: $\pi_0 = (1-p)\pi_0 + (1-p)\pi_1, \pi_1 = (1-p)\pi_2, \pi_2 = (1-p)\pi_3, \pi_3 = p\pi_0 + p\pi_1 + p\pi_2 + p\pi_3, \pi_0 + \pi_1 + \pi_2 + \pi_3 = 1$. Solving yields $\pi_0 = 1 - 3p + 3p^2 - p^3, \pi_1 = p - 2p^2 + p^3, \pi_2 = p - p^2$ and $\pi_3 = p$. Hence the probability that the buffer contains two frames in the long run is $p - p^2$.
- (b) $p - p^2$ corresponds to a downward opening parabola, the top of which lies at $p = \frac{1}{2}$. The corresponding maximal probability equals $\frac{1}{4}$.

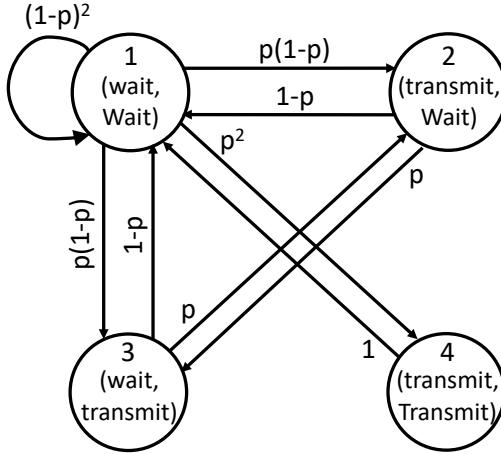


Figure B.18: Transition diagram of time-slotted medium with two packet generators

Exercise B.30 (Packet generator - generated load).

- (a) We have to compute the long-run expected fraction of visits to state 2. For $p = 1$, the chain is a non-ergodic unichain. For $0 \leq p < 1$, the chain is an ergodic unichain. Hence for any value of p , the chain is a unichain and therefore the balance equations have a unique solution. The balance equations are given by $\pi_1 = (1 - p)\pi_1 + \pi_2$, $\pi_2 = p\pi_1$, $\pi_1 + \pi_2 = 1$. Solving yields $\pi_1 = \frac{1}{1+p}$ and $\pi_2 = \frac{p}{1+p}$. Hence the generated load is $\frac{p}{1+p}$ packets per time slot.
- (b) The minimal load is 0 and is obtained for $p = 0$. The maximal load is $\frac{1}{2}$ and is obtained for $p = 1$.

Exercise B.31 (Expected fraction of time spent in a state equals reciprocal of expected return time).

- (a) From the transition diagram, it is immediately clear that all states in this chain communicate. Hence the chain has a single class of recurrent states and is thus irreducible. To understand why this chain is not ergodic, consider state 1. The length of the paths from state 1 to 1 are 4, 6, 8, \dots and hence $d(1) = 2$. State 1 is thus periodic and so are all the other states.
- (b) The long-run expected fraction of time the chain spends in state 1 equals $\pi_1^{(\infty)}$, where $\pi^{(\infty)}$ is the unique solution to the balance equations: $\pi_1 = \pi_4$, $\pi_2 = \pi_1 + \frac{5}{6}\pi_3$, $\pi_3 = \pi_2$, $\pi_4 = \frac{1}{6}\pi_3$ and $\pi_1 + \pi_2 + \pi_3 + \pi_4 = 1$. Solving yields $\pi^{(\infty)} = [\frac{1}{14}, \frac{3}{7}, \frac{3}{7}, \frac{1}{14}]$. Hence $\pi_1^{(\infty)} = \frac{1}{14}$. The expected return time to state 1 is given by $\frac{f_{11}}{f_{11}}$, where reward r assigns the value 1 to each state. Since state 1 is recurrent, $f_{11} = 1$. To compute f_{11}^r we solve the system of equations in (B.39): $x_1 = 1 + x_2$, $x_2 = 1 + x_3$, $x_3 = 1 + \frac{5}{6}x_2 + \frac{1}{6}x_4$, $x_4 = 1$. Solving yields $x_1 = 14$, $x_2 = 13$, $x_3 = 12$ and $x_4 = 1$. Hence $f_{11}^r = 14 = \frac{1}{\pi_1^{(\infty)}}$.

Exercise B.32 (Composition of two parallel packet generators - generated load).

- (a) The system can be modelled by a product of two traffic generators, see Figure B.18. State 1, for instance, indicates that both traffic generators are in their wait states and state 2 indicates that one traffic generator is transmitting while the other is waiting. Transitions refer to the logic AND of the corresponding transitions of the generators. E.g the probability of the transition from state 1 to state 2 is the product of probabilities that one generator transits from the wait to the transmit state (p) and the other remains in the wait state ($1 - p$).
- (b) We define a reward r as follows: $r(1) = 0, r(4) = 0, r(2) = 1, r(3) = 1$. The utilization is then given by long-run expected (average) reward is given by $\pi^{(\infty)}r^T = \pi_2^{(\infty)} + \pi_3^{(\infty)}$. We have to distinguish two cases:
- (1) For $0 \leq p < 1$ we are dealing with an ergodic unichain. This means that the limit distribution $\pi^{(\infty)}$ exists, is independent of the initial distribution, and is given by the unique solution to the balance equations. These equations are:
 $\pi_1 = (1-p)^2\pi_1 + (1-p)\pi_2 + (1-p)\pi_3 + \pi_4, \pi_2 = p(1-p)\pi_1 + p\pi_3, \pi_3 = p(1-p)\pi_1 + p\pi_2,$
 $\pi_4 = p^2\pi_1$ and $\pi_1 + \pi_2 + \pi_3 + \pi_4 = 1$. Solving yields $\pi_1 = \frac{1}{(1+p)^2}, \pi_2 = \frac{p}{(1+p)^2}, \pi_3 = \frac{p}{(1+p)^2}$ and $\pi_4 = \frac{p^2}{(1+p)^2}$. Thus $\pi^{(\infty)}r^T = \frac{2p}{(1+p)^2}$.
 - (2) For $p = 1$ the chain is a non-ergodic non-unichain; it consists of two recurrent periodic classes $\{1, 4\}$ and $\{2, 3\}$. Thus the normal limit $\pi^{(\infty)}$ does not exist and neither does the limit of the probability matrix. Further $\pi^{(\infty)}$ will depend on the initial distribution in this case. It is easy to find out that

$$P^\infty = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

so that $\pi^{(\infty)} = \pi^{(0)}P^\infty = [\frac{1}{2}\pi_1^{(0)} + \frac{1}{2}\pi_4^{(0)}, \frac{1}{2}\pi_2^{(0)} + \frac{1}{2}\pi_3^{(0)}, \frac{1}{2}\pi_2^{(0)} + \frac{1}{2}\pi_3^{(0)}, \frac{1}{2}\pi_1^{(0)} + \frac{1}{2}\pi_4^{(0)}]$.
Thus $\pi^{(\infty)}r^T = \pi_2^{(\infty)} + \pi_3^{(\infty)} = \pi_2^{(0)} + \pi_3^{(0)}$.

- (c) We have to distinguish the same two cases again:

- (1) For $0 \leq p < 1$ we have to find the optimum of $\frac{2p}{(1+p)^2}$. $\frac{2p}{(1+p)^2}$ is increasing in p , so the optimal value is obtained when p approaches 1. The optimal utilization is $\frac{1}{2}$. Apparently, even though both generators can produce a load of 50%, collisions prevent the system from reaching a utilization beyond 50%.
- (2) For $p = 1$, we have to find the optimum of $\pi_2^{(0)} + \pi_3^{(0)}$. The optimum is 1 in case the sum of the initial probabilities of states 2 and 3 equals 1. This means that the packets generators are never in the same states and thus collisions are always avoided.

Exercise B.33 (Computer system - throughput).

Exercise B.33 (Computer system - throughput).

- (a) The computer system can be modeled with three states A , B and C . Transition probabilities are $P_{AA} = 1 - p$, $P_{AB} = p$, $P_{BA} = 1 - p$, $P_{BC} = p$ and $P_{CA} = 1$.
- (b) For $0 \leq p < 1$ the Markov chain is an ergodic unichain and for $p = 1$ the Markov chain is a non-ergodic unichain. In any case, we are dealing with a unichain for which the Cesàro limiting distribution exists. It can be found by solving the balance equations. Solving yields $\pi_A = \frac{1}{1+p+p^2}$, $\pi_B = \frac{p}{1+p+p^2}$ and $\pi_C = \frac{p^2}{1+p+p^2}$.
- (c) We can model the execution times by assigning rewards to the states. In states A , B and C rewards 1, $3 + 3p$ and $2p$ are obtained respectively. The long-run expected average reward obtained is then $\pi_A + (3 + 3p)\pi_B + 2p\pi_C = \frac{1+3p+3p^2+2p^3}{1+p+p^2} = 2p + 1$. This means that the expected average time per execution equals $2p + 1$ seconds. Hence the throughput is $\frac{1}{2p+1}$ tasks per second.
- (d) Expression $\frac{1}{2p+1}$ is decreasing in p and is thus minimal for $p = 1$. For $p = 1$ the throughput is $\frac{1}{3}$ tasks per second.

Exercise B.34 (Throughput of an Ethernetwork - confidence levels versus error bounds).

- (a) For $\gamma = 0.10$, c equals $F_{normal}^{-1}((1.0 + 0.10)/2.0)$ which is approximately 0.126. The confidence interval we obtain is $[0.48, 0.51]$. The absolute and relative error bounds are 0.020 and 0.041 respectively, which are reasonable in practice.
- (b) Although the estimation yields reasonable error bounds, the confidence is very low. Only 10% of confidence intervals that are obtained via simulations of length 10 contain the true value of μ . The error bounds are computed based on the assumption that μ is part of the interval, but in this case most likely it is not.

Exercise B.35 (Throughput of an Ethernetwork - required length of simulation sequence). Since the expected value μ is known, we can compute the variance as $\sigma^2 = Var(Y_n) = E((Y_n - \mu)^2) = E(Y_n^2 - 2\mu Y_n + \mu^2) = E(Y_n^2) - 2\mu^2 + \mu^2 = E(Y_n^2) - \mu^2$. Now $E(Y_n^2) = 1 \cdot \mu + 0 \cdot (1 - \mu) = \mu$. Hence $\sigma^2 = \mu - \mu^2 = \mu(1 - \mu)$ and $\sigma = \sqrt{\mu(1 - \mu)}$. For $\mu = 0.6$ we thus obtain $\sigma \approx 0.775$. Now instead of using the interval estimator S_M of σ in (B.62) we can use σ itself. This leads to an absolute error bound $\frac{c\sigma}{\sqrt{M}}$. With a confidence level of 95%, $c\sigma = 1.96 * 0.775 \approx 1.52$. Hence we need $\frac{1.52}{\sqrt{M}} \leq 0.001$. This inequality holds for $M \geq 2210400$, so we require a simulation run of more than 2 million observations.

Exercise B.36 (Interval estimator of expected value). We know from Equation B.61 that

$$P(-c \leq \sqrt{M} \frac{\frac{1}{M} \sum_{n=0}^{M-1} Y_i - \mu}{S_M} \leq c) \approx \gamma$$

Hence

$$P(-cS_M \leq \sqrt{M}(\frac{1}{M} \sum_{n=0}^{M-1} Y_i - \mu) \leq cS_M) \approx \gamma$$

Thus

$$P(\frac{-cS_M}{\sqrt{M}} \leq \frac{1}{M} \sum_{n=0}^{M-1} Y_i - \mu \leq \frac{cS_M}{\sqrt{M}}) \approx \gamma$$

So

$$P(\frac{-cS_M}{\sqrt{M}} - \frac{1}{M} \sum_{n=0}^{M-1} Y_i \leq -\mu \leq \frac{cS_M}{\sqrt{M}} - \frac{1}{M} \sum_{n=0}^{M-1} Y_i) \approx \gamma$$

Therefore

$$P(\frac{1}{M} \sum_{n=0}^{M-1} Y_i + \frac{cS_M}{\sqrt{M}} \geq \mu \geq \frac{1}{M} \sum_{n=0}^{M-1} Y_i - \frac{cS_M}{\sqrt{M}}) \approx \gamma$$

And thus

$$P(\frac{1}{M} \sum_{n=0}^{M-1} Y_i - \frac{cS_M}{\sqrt{M}} \leq \mu \leq \frac{1}{M} \sum_{n=0}^{M-1} Y_i + \frac{cS_M}{\sqrt{M}}) \approx \gamma$$

Rewriting yields the result:

$$P(\mu \in [\frac{1}{M} \sum_{n=0}^{M-1} Y_n - \frac{cS_M}{\sqrt{M}}, \frac{1}{M} \sum_{n=0}^{M-1} Y_n + \frac{cS_M}{\sqrt{M}}]) \approx \gamma$$

Exercise B.37 (Confidence interval interpretation). No, the claim is false. μ is either in the interval, or it is not. It can not be in the interval with probability 0.95. A correct claim would have been: 'The confidence level that μ is contained in the interval [1.31, 4.25]' is 95%. This means that about 95% of all the intervals that could have been obtained will include value μ .

Exercise B.38 (Standard deviation - point estimator). For this we have to show that for almost any sequence of realizations y_0, y_1, \dots we have

$$\lim_{M \rightarrow \infty} \sqrt{\frac{1}{M} \sum_{n=0}^{M-1} (y_n - \frac{1}{M} \sum_{m=0}^{M-1} y_m)^2} \rightarrow \sigma$$

From the strong law of large numbers we know that $\lim_{M \rightarrow \infty} \frac{1}{M} \sum_{m=0}^{M-1} y_m \rightarrow \mu (= E(Y_n))$. Hence $\lim_{M \rightarrow \infty} s_M = \lim_{M \rightarrow \infty} \sqrt{\frac{1}{M} \sum_{n=0}^{M-1} (y_n - \mu)^2}$. Now $(Y_0 - \mu)^2, (Y_1 - \mu)^2, \dots$ is a sequence of identically distributed random variables with mean $\sigma^2 = ((Var)(Y_n))$. Hence by the strong law of large numbers we have $\lim_{M \rightarrow \infty} \frac{1}{M} \sum_{n=0}^{M-1} (y_n - \mu)^2 = \sigma^2$. Therefore $\lim_{M \rightarrow \infty} \sqrt{\frac{1}{M} \sum_{n=0}^{M-1} (y_n - \mu)^2} = \sqrt{\sigma^2} = \sigma$.

Exercise B.39 (Estimation transient distributions). To apply the basic theory of estimation we have to define a sequence Y_0, Y_1, \dots of independent identically distributed variables such that $E(Y_n) = \pi_i^{(K)}$. Now $\pi_i^{(K)}$ is not defined as an expected value. We do know however that $E(r(X_K)) = \pi^{(K)} \cdot r^T$ (see (B.5))). If we now define r as $r(k) = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{if } k \neq i \end{cases}$, then $\pi_i^{(K)} = \pi^{(K)} \cdot r^T = E(r(X_K))$. The way to estimate $E(r(X_K))$ is already explained in Example B.21.

Exercise B.40 (Gambler's ruin - expected reward estimation).

- (a) When the point estimation s_M is replaced by σ in Equation B.64, the absolute error bound is given by $\frac{c\sigma}{\sqrt{M}}$. For a 95% confidence interval $c \approx 1.96$. Further $\sigma^2 = \text{Var}(r(X_2)) = \{ \text{see also Exercise B.35} \} E(r(X_2)^2) - E(r(X_2))^2$. Now $E(r(X_2)) = 150$ and $\pi^{(2)} = [\frac{3}{8}, \frac{1}{8}, \frac{1}{8}, \frac{3}{8}]$. Hence $\sigma^2 = E(r(X_2)^2) - 150^2 = \frac{3}{8} \times 0^2 + \frac{1}{8} \times 100^2 + \frac{1}{8} \times 200^2 + \frac{3}{8} \times 300^2 = 40,000 - 22,500 = 17,500$ and thus $\sigma \approx 132$. For the absolute error bound to be 2.5 we thus need $\frac{1.96 \times 132}{\sqrt{M}} = 2.5$. This holds for $M \approx 10,710$ realizations.
- (b) Simulation in the CMWB could deliver estimation 149.0943 with confidence interval [146.5805, 151.6081], corresponding to an absolute error bound of 2.5138. The small difference in the bound is due to the fact that the simulation uses an estimation of σ and not the true value.

Exercise B.41 (Gambler's ruin - converge rate central limit theorem).

- (a) $\pi^{(10)} = [0.4995, 0.0005, 0.0005, 0.4995]$.
- (b) The simulation could deliver distribution [0.4947, 0.0004, 0.0002, 0.5047]. For each element in the vector we can estimate the absolute error bound. The first element resulted from the estimation of $E(r(X_{10}))$, where reward $r = [1, 0, 0, 0]$. The absolute error bound for this element is given by $\frac{c\sigma}{\sqrt{M}}$, where $c = 1.96$, $\sigma = \sqrt{E(r(X_{10})^2) - E(r(X_{10}))^2} = \sqrt{0.4995 - 0.4995^2} = 0.5000$ and $M = 10,000$. Hence $\frac{c\sigma}{\sqrt{M}} \approx 0.01$. The same error bound holds for the fourth element. In a similar way we can derive an error bound of about 0.00044 for the second and third elements. Hence the maximal error bound is about 0.01.
- (c) When the absolute error bound of 0.0001 is used as the stopping criterion in the CMWB, the simulation will most likely terminate after 30 realization with an estimated expected reward 0.0000, confidence interval [0.0000, 0.0000] and with an absolute error bound 0.0000. How can this happen? The simulator uses the rule of thumb that the distribution $\sum_{n=0}^{M-1} Y_n$ is approximately normal for $M \geq 30$ (see Equation B.60) and therefore simulates at least 30 paths. In this particular case, where each Y_n has a Bernoulli distribution with parameter 0.0005, the error between the normal

distribution and binomial distribution of $\sum_{n=0}^{M-1} Y_n$ is still quite large¹². Now point estimator $\frac{1}{M} \sum_{n=0}^{M-1} Y_n$ is used to estimate $E(r(X_{10}))$, since it converges almost surely to this value. However, the probability that $\frac{1}{30} \sum_{n=0}^{29} Y_n$ takes a value between 0.0004 and 0.0006 is 0, while the probability that it takes value 0 exceeds 0.98¹³. So after precisely 30 paths, the odds are high that only realizations with value 0 have been created, yielding the results stated above. To estimate $E(r(X_{10}))$ with an error bound of 0.0001, we could use the 'Maximum number of realizations' option in CMWB and set it to a value of about 200,000.

Exercise B.42 (Estimation hitting probability - impact of maximal path length).

- (a) Random variable Y is defined such that Y takes value 1 if state 3 is hit from state 1 in at most 3 transitions and 0 otherwise. Two of such path that hit state 3 exist, namely 1, 2, 3 with probability $\frac{1}{6}$ and 1, 2, 2, 3 with probability $\frac{1}{9}$. Hence $E(Y) = \frac{5}{18}$ and thus $\frac{1}{M} \sum_{n=0}^{M-1} Y_n$ converges almost surely to $\frac{5}{18}$. Notice that the point estimation is an underestimation of f_{13} .
- (b) The relative error bound is given by $\frac{cs_M/\sqrt{M}}{\hat{\mu}-cs_M/\sqrt{M}}$ (see Equation B.65). Here point estimations $\hat{\mu}$ and s_M can be replaced by $E(Y) \approx 0.28$ and $\sqrt{Var(Y)} = \sqrt{\frac{5}{18} - \frac{5^2}{18^2}} = \sqrt{\frac{65}{324}} \approx 0.45$ respectively. For $\gamma = 0.90$, we get $c \approx 1.65$. To obtain a relative error bound of 0.01 we thus require $\frac{1.65 \times 0.45 / \sqrt{M}}{0.28 - 1.65 \times 0.45 / \sqrt{M}} \leq 0.01$. This holds for $M \geq 71734$.

Exercise B.43 (Estimation expected cumulative reward until hit). We have to define a sequence of independent identically distributed variables Y_0, Y_1, \dots such that $E(Y_n) = \frac{f_{ij}}{f_{ii}}$. Assume we would define variable Y as the cumulative reward earned if state j is hit from i in one or more steps and 0 otherwise. Then $E(Y)$ takes value f_{ij}^r and not $\frac{f_{ij}^r}{f_{ii}}$ since all paths are taken into account, including those that do not hit state j . To make sure that $E(Y) = \frac{f_{ij}}{f_{ii}}$, we define Y only for those paths that actually hit j ¹⁴. We then let each Y_n be an independent copy of Y . A realization y_n is obtained by (re-)starting the Markov chain in state i , after which a path (respecting the transition probabilities) is generated. If the path hits state j , y_n equals the cumulative reward obtained along this path. In case state j is not hit, the path is simply discarded. Similar to Example B.22, this is decided when the length of the generated path exceeds some pre-defined number K .

Exercise B.44 (Rover in a maze - estimation escape probability). A good approach is to start experimenting with a confidence level of 95% and a relative error bound of 1%. To

¹²Using the Berry-Esséen theorem, it can be shown that this error is bounded by $\frac{C(p^2 + (1-p)^2)}{\sqrt{np(1-p)}}$, where C is a constant less than 0.7655 and p is the parameter of the Bernoulli distribution. For $p = 0.0005$ the error is thus bounded by 0.0936, a large number compared to 0.0005.

¹³This can be shown using a tool like Mathematica.

¹⁴In fact this boils down to computing the so-called conditional expected value, i.e. the expected value conditional on the fact that j is hit.

make sure that the simulation will terminate on the specified relative error bound, a sufficiently large number of paths to generate should be specified. In this particular case 1000,000 paths will do. Since we are estimating a hitting probability, the maximal path length should be specified as well. The bigger the maximal path length, the better the estimation will be. A good way to determine an appropriate maximal path lenght is to start with a modest length, say 10, and incrementally increase the lenght by doubling it. One can continue until the estimated values are stablizing, e.g. by checking whether the most recent estimation is in the confidence interval corresponding to the prior estimation. For the rover example, the results could look like:

| maximal path length | point estimation | interval estimation |
|---------------------|------------------|---------------------|
| 10 | 0.1673 | [0.1657, 0.1690] |
| 20 | 0.2338 | [0.2314, 0.2361] |
| 40 | 0.2818 | [0.2790, 0.2846] |
| 80 | 0.2891 | [0.2863, 0.2920] |
| 160 | 0.2879 | [0.2851, 0.2908] |

As a resulting point estimation we thus obtain 0.2879, which is very close to the computed value of $\frac{11}{38} \approx 0.2895$.

Exercise B.45 (Absolute error bound long-run expected average reward).

- (a) The point estimation for μ is given by $\hat{\mu} = \frac{\sum_{n=0}^{M-1} r_n}{\sum_{n=0}^{M-1} l_n}$ and the confidence interval is given by $[\hat{\mu} - \frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n}, \hat{\mu} + \frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n}]$. Notice that $\hat{\mu}$ lies in the middle of this interval. To determine an error bound, we have to assume that μ is in the interval. It is then easy to see that $|\mu - \hat{\mu}| \leq \frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n}$.
- (b) Value c is a constant and s_M converges almost surely to $\sqrt{Var(Y_n)}$. $\frac{1}{M} \sum_{n=0}^{M-1} l_n$ converges almost surely to $E(L_n) = \frac{1}{\pi_{ir}^{(\infty)}}$. Therefore the bound converges almost surely to 0.

Exercise B.46 (Long-run expected average reward - confidence interval). The sequence contains the following segments:

- 1, 1: this initial segment is discarded;
- 2, 3, 3: this is the zeroed cycle through the recurrent state; $l_0 = 3$ and $r_0 = 8$;
- 2, 4: this is the first cycle through the recurrent state; $l_1 = 2$ and $r_1 = 6$;
- 2, 4: this is the second cycle through the recurrent state; $l_2 = 2$ and $r_2 = 6$;
- 2, 3, 3, 3: this is the third cycle through the recurrent state; $l_3 = 4$ and $r_3 = 11$;
- 2, 3: this is the start of the fourth cycle; it can not be determined whether it is complete already and is therefore not (yet) taken into account.

We therefore have $M = 4$, $\hat{\mu} = \frac{\sum_{n=0}^3 r_i}{\sum_{n=0}^3 l_i} = \frac{31}{11} \approx 2.82$, $s_4 = \sqrt{\frac{1}{4} \sum_{n=0}^3 (r_n - \frac{\sum_{m=0}^3 r_m}{\sum_{m=0}^3 l_m} l_n)^2} \approx 0.37$

and $\frac{1}{4} \sum_{n=0}^3 l_n = 2.75$. Further for $\gamma = 0.95$, $c = 1.96$. The confidence interval is therefore $[\hat{\mu} - \frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n}, \hat{\mu} + \frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n}] \approx [2.69, 2.95]$ with an absolute error bound $\frac{cs_M}{\sqrt{M} \frac{1}{M} \sum_{n=0}^{M-1} l_n} \approx 0.13$.

Exercise B.47 (Expected return time versus long-run expected average reward). The chain is a non-ergodic unichain. Solving the balance equations yields $\pi^{(\infty)} = [\frac{1}{14}, \frac{3}{7}, \frac{3}{7}, \frac{1}{14}]$ and thus $\pi_2^{(\infty)} = \frac{3}{7}$. Hence $\pi^{(\infty)} \cdot r^T = \frac{83}{14}$. Solving the equations to compute the expected reward until hitting state 2 yields $f_{22}^r = \frac{83}{6}$. The result follows from the fact that $\frac{83}{6} = \frac{83}{14}/\frac{3}{7}$.

Exercise B.48 (Interval estimator long-run expected average reward). From Equation B.67 we know

$$P(-c \leq \frac{\sum_{n=0}^{M-1} Y_n}{\sqrt{MS_M}} \leq c) \approx \gamma$$

Replacing Y_n by $R_n - \mu L_n$ yields:

$$P(-c \leq \frac{\sum_{n=0}^{M-1} R_n - \mu \sum_{n=0}^{M-1} L_n}{\sqrt{MS_M}} \leq c) \approx \gamma$$

Multiplying by $\sqrt{MS_M}$ and subtracting $\sum_{n=0}^{M-1} R_n$ gives:

$$P(-c\sqrt{MS_M} - \sum_{n=0}^{M-1} R_n \leq -\mu \sum_{n=0}^{M-1} L_n \leq c\sqrt{MS_M} - \sum_{n=0}^{M-1} R_n) \approx \gamma$$

Multiplying by -1 and dividing by $\sum_{n=0}^{M-1} L_n$ yields:

$$P(\frac{\sum_{n=0}^{M-1} R_n}{\sum_{n=0}^{M-1} L_n} - \frac{c\sqrt{MS_M}}{\sum_{n=0}^{M-1} L_n} \leq \mu \leq \frac{\sum_{n=0}^{M-1} R_n}{\sum_{n=0}^{M-1} L_n} + \frac{c\sqrt{MS_M}}{\sum_{n=0}^{M-1} L_n}) \approx \gamma$$

The results follows by dividing by \sqrt{M} .

Exercise B.49 (Estimation long-run expected fraction of time spent in a state). Because the chain is a unichain, all rows in P^∞ are equal to $\pi^{(\infty)}$. Hence $P_{ij}^\infty = \pi_j^{(\infty)}$. If we now define reward r as $r(k) = \begin{cases} 1 & \text{if } k = j \\ 0 & \text{if } k \neq j \end{cases}$, then $\pi_j^{(\infty)} = \pi^{(\infty)} \cdot r^T$. Expressions for the point estimator and interval estimator of $\pi^{(\infty)} \cdot r^T$ are already known and given in Equations B.66 and B.68 respectively.

Exercise B.50 (Estimation Cezàro limiting distribution of a non-unichain). The simulation never converges to the limit distribution. The reason is that the chain is a non-unichain with two classes of recurrent states. The simulator generates a single path through this chain. The first recurrent state it encounters on this path will be used as the state

for generating cycles. In the example this is state 4, state 5 or state 6. In case state 4 or state 5 is selected, recurrent class $\{4, 5\}$ is entered and the estimation will converge to $[0, 0, 0, \frac{1}{2}, \frac{1}{2}, 0, 0, 0]$. This vector corresponds to rows 4 and 5 of the Cesàro limiting matrix in Example B.18. Otherwise recurrent class $\{6, 7, 8\}$ is entered and the estimation will converge to $[0, 0, 0, 0, 0, \frac{4}{9}, \frac{2}{9}, \frac{1}{3}]$. This vector corresponds to rows 6, 7 and 8 of the limiting matrix.

Exercise B.51 (Video application - long-run average buffer occupancy). To compute the occupancy level, each state i of the Markov chain can be assigned a reward i . The occupancy level is then given by the long-run expected reward. The computed value equals 2.125. Simulation with a relative error bound of 0.01 could yield point estimation 2.1339 and interval estimation [2.1127, 2.1550], requiring 9390 simulation cycles.

C

Max-Plus Linear Systems, Dataflow Models and Performance Analysis

Introduction

Languages and automata describe behavior of systems in terms of the possible sequences of actions or activities that they may exhibit. Markov Chains and, in general, stochastic and statistical models, introduce a quantitative sense of the *likelihood* of such sequences to occur. In this section we turn our attention to *performance models* that innately include a quantitative measure of time. This allows us to determine the measure of time it takes to complete certain actions. This way we can determine *performance qualities* such as *makespan*, *latency* or *throughput*.

We break the behavior of a system down into independent actions that each take their own, fixed, amount of time. To construct meaningful models from such actions we assume that actions may proceed concurrently, in general, but there may also be dependencies between activities that constrain their order of occurrence and therefore the progress that can be made. We further allow such models to have dependencies on input events, and to be able to produce events to their environments to model systems that produce outputs. We introduce a concrete model in which such systems can be expressed, namely, *Timed Synchronous Dataflow* models. We refer to them, in short, as SDF models. We show that, although SDF models are deterministic, they can effectively be used for worst-case performance analysis of systems that exhibit timing variations.

We introduce methods to study different aspects of their performance. For this we discuss their behavior (*semantics*). The mathematical framework in which to express the behavior of SDF graphs is *max-plus algebra*. We introduce this algebra and we study the essential properties of the class of max-plus-linear systems and how to use those properties to analyze SDF models. We discuss a particular, generic representation of max-plus-linear systems called the state-space representation and develop results to study throughput, latency and stability.

More information about the topics in this section are found in the cited references. More information on dataflow models can be found in [5, 12]. Background on max-plus algebra and max-plus-linear systems can be found in [14, 2, 8, 9]. Modelling of manufacturing systems

and cyber-physical systems is discussed in [4]. More about the modelling of railroad systems can be found in [13]. The notes in this section are partially based on these works.

Learning objectives.

At the end of this module, the student will be able to

- **understand** max-plus discrete-time time signals vs. the classical time view on system operation;
- **understand** impulse response and *compute* the convolution of max-plus signals
- **understand** the Max-Plus Linear Systems (MPLS) state-space equations;
- **understand** the properties of linearity, superposition and stability;
- **understand** (open) single-rate dataflow models;
- **create** single-rate dataflow models;
- **convert** single-rate dataflow models to MPLS;
- **create** the state-space equations of an MPLS;
- **analyze** maximum eigenvalue and eigenvectors of a max-plus matrix
- **analyze** throughput, latency, and stability;
- **apply** the superposition property.

C.1 Timed Synchronous Dataflow Models

Dataflow is an abstract mathematical model that can be used to model repetitive deterministic operations on streams of products, or streams of data items, often including the resources on which they are realized. They are used, for instance, to model applications on processor platforms, to model manufacturing systems, or train schedules.

Definition C.1 (Timed Dataflow Graph). A timed dataflow graph consists of

- a finite set A of *actors*;
- *firing durations* $e : A \rightarrow \mathbb{R}_{\geq 0}$;
- a finite set $C \subseteq A \times \mathbb{N} \times A$ of *channels*;
- a finite set I of *inputs*, with *input dependencies* $d_I : I \rightarrow A$;
- a finite set O of *outputs*, with *output dependencies* $d_O : O \rightarrow A$;

Actors from A represent the activities in the model. An actor can *fire*, which means that it executes its activity. Actors can fire repetitively and indefinitely, unless there are constraints that prevent it. The firings of an actor have a fixed duration given by the function e and multiple firings of the same actor can be ongoing simultaneously. A channel (a_1, n, a_2) is a dependency of the firings of actor a_2 on the firings of the actor a_1 . The firing number $k + n$ of actor a_2 cannot start before firing number k of a_1 is complete. There is no dependency for the first n firings of a_2 . An input dependency $d_I(i) = a$ means that firing k of actor a depends on the presence of input number k on input i . Output dependency $d_O(o) = a$ means that output number k on output o is produced by firing k of actor a .

Example C.1 (Dataflow Example). Figure C.1 shows an example of an SDF model. The circles in the figure are the actors. In a representation of an SDF graph, we often give actors a name. In Figure C.1, the names are written inside the actors A, B, C. The durations of the firings are also written inside the actor. For instance, the firings of actor C take 4 time units each. Arrows between actors represent the channels. For example, the channel from A to C creates the constraint that firing number k of actor C cannot start before firing number k of A is complete. This can be described operationally as if actor A *produces* an output *token* onto the channel every time it completes a firing and actor C *consumes* a token when it starts its firing. We often use this operational view when it is more convenient. The channel from B to A has an *initial token*, even before B has completed any firings. Without it a cyclic dependency would prevent actors A and B from ever firing.

The graph has one input i . The input dependency is shown by an arrow from the input to the actor that depends on it. The graph has one output o . The output dependency is shown by an arrow from the actor that produces it to the output.

If we assume that the first firings start at time 0 and further all actor firings start as soon as allowed by the constraints, then behavior including the first four firings of each actor is shown in Figure C.2 in the form of a Gantt chart. The horizontal axis is a time line. On the first row, the arrival of the four inputs is shown with dots at the points in time when they arrive. Actor A is the first to fire when the first input, on which it depends, arrives. Actors B and C follow with their first firings when A completes its first firing. Observe that the firings of B and C occur in parallel. Observe also that the firings of actor C overlap. For instance, its second firings starts at time 5, while its first firing only ends at time 6. The last row shows the outputs produced by actor C onto output o . They coincide with the completions of the firings of C.

Example C.2 (Dataflow Model of a Manufacturing System). Figure C.3 shows a simple manufacturing system. It takes two kinds of objects on the input conveyor belt. We call those types of objects ‘bottoms’ (shown as red objects) and ‘tops’ (shown as green objects) and the purpose of the system is to combine them into a combined product by placing the tops on the bottoms. The bottoms and tops are fed into the

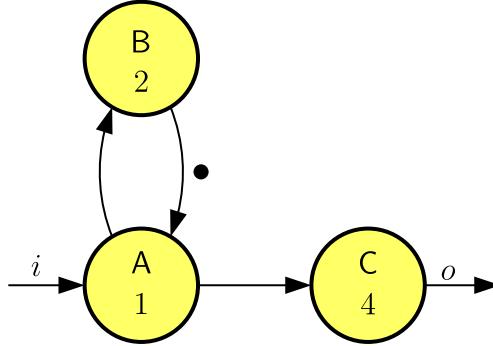


Figure C.1: Example of a timed dataflow model.

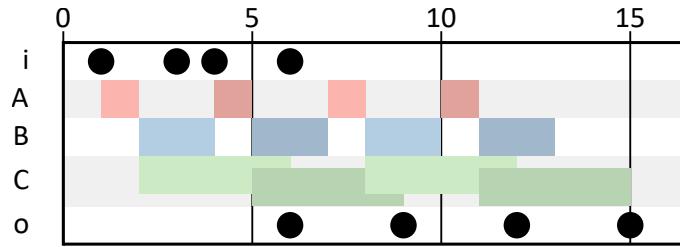


Figure C.2: Gantt chart with the behavior of a dataflow model.

system alternatingly, starting with a top. Tops are transported by conveyor belts to a *pick-and-place unit*. Bottoms are transported on the conveyor belt to a rotating *indexing table*, which will rotate it towards the pick-and-place unit that will place the top on top of it. The combined product is transported by the indexing table to another conveyor belt, which will move the object to the system output. The indexing table always simultaneously rotates a new bottom to the pick-and-place unit and the combined product to the belt. Therefore, at startup of the system, a bottom part is initially placed on the indexing table at the pick-and-place unit.

Figure C.4 shows a dataflow model of this manufacturing system. It has two inputs, in_{bottom} and in_{top} representing the arrivals of bottoms and tops, respectively. Tops and bottoms are admitted onto the conveyor belt alternatingly. This is modeled by the cycle of actors InT — $Pass2$ — InB — $Pass1$. InB fires when a new bottom is admitted and InT when a top is admitted. This happens when the object is available and the belt is clear. Actors $Pass1$ and $Pass2$ model the time it takes after admission of an object to clear the belt for another object. The initial token on the cycle ensures that the first object admitted is a top.

The sequence of actors at the top of the figure follows the operations applied to the bottoms. Actor BB models its transport to the indexing table. Rot models its rotating on the table towards the pick-and-place unit. PaP models the pick-and-place unit placing a top on it. $Exit$ models the final transport towards the output including the rotation on the index table and the transport on the belt to the output. The BT actor models the transport of the top on the belt to the pick-and-place unit, where it is

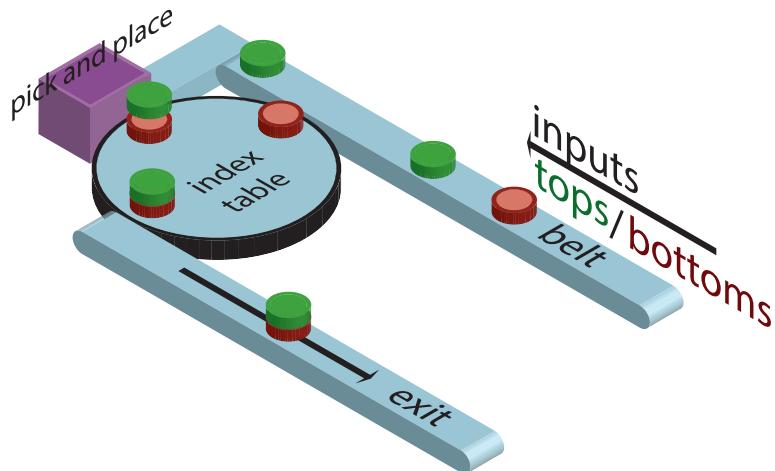


Figure C.3: Model of a manufacturing system.

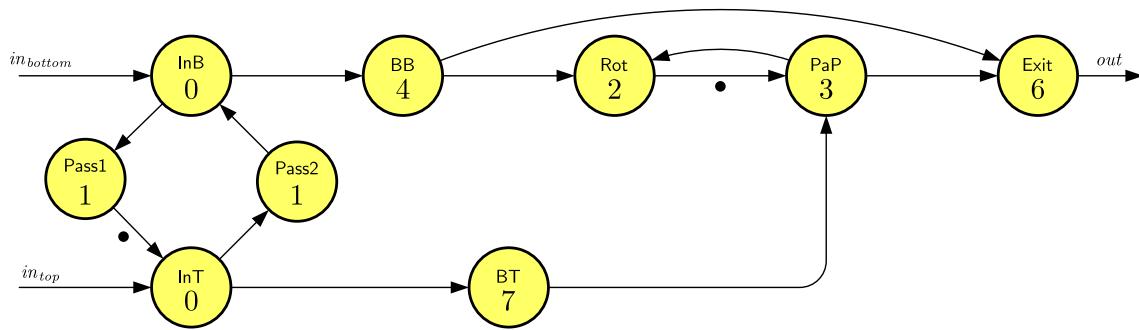


Figure C.4: Dataflow model of a manufacturing system.

combined with a bottom.

Most dependencies in the graph are straightforward. The initial token on the channel from Rot to PaP represents the bottom part that is initially placed in the system. The dependency from PaP to Rot ensures that the indexing table does not rotate until the pick-and-place operation is complete. The dependency from BB to Exit ensures that the second rotation is correctly modelled. It only starts when a new bottom part is ready to move onto the indexing table.

Figure C.5 shows a Gantt chart of the behavior of the manufacturing system. Note that the arrivals of tops and bottoms are arbitrary, but early enough to not impact the schedule. Observe the cycle of firings of InT, Pass2, InB and Pass1 as the objects are admitted to the system. The firings of the BT and BB actors overlap, as multiple objects are transported on the same conveyor belt simultaneously. Rot and Exit always start simultaneously as they both capture the rotating of the indexing table. Completed products appear at the output starting from time 16 periodically with 5 time units in between. It appears that the critical path for producing the outputs is formed by the alternating PaP and Rot actors.

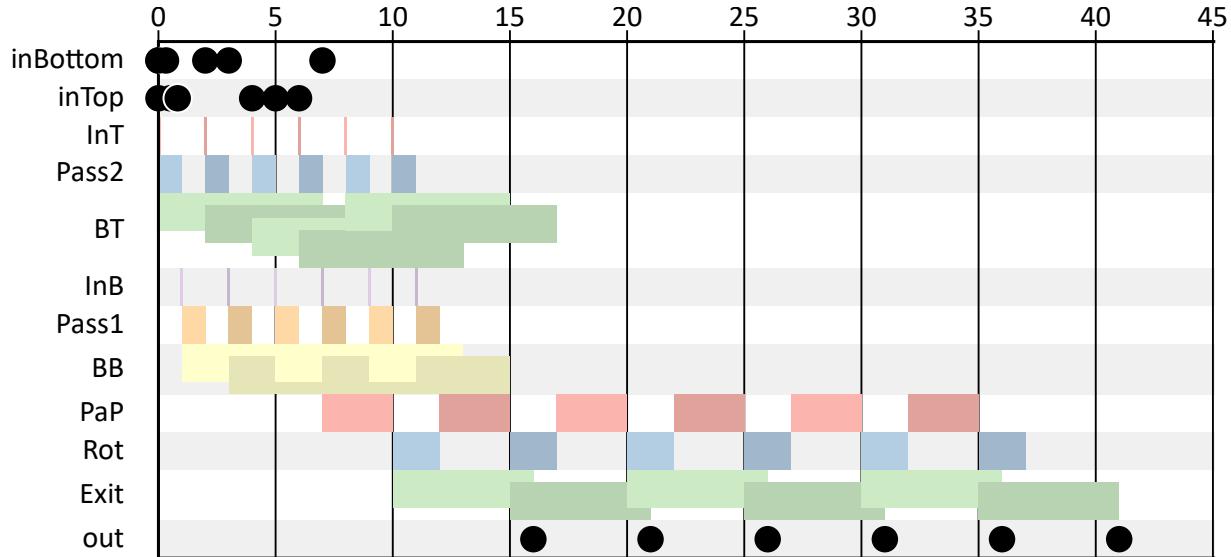
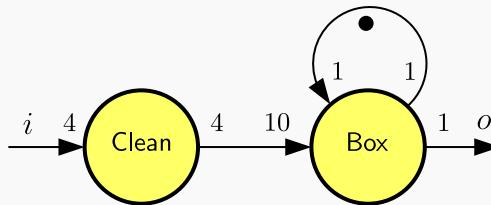


Figure C.5: Gantt chart of the manufacturing system.

The timed dataflow models we have introduced are sometimes also called *single-rate* dataflow models. There are also *multi-rate* timed dataflow models. In a multi-rate dataflow graph, actor firings may produce and/or consume multiple tokens with every firing. This rate must however be *constant*, i.e., every firing of the same actor always produces and consumes the same quantity of tokens. The number $c \in \mathbb{N}$, $c > 0$ of tokens it consumes from an input channel is called the *consumption rate* and the number $p \in \mathbb{N}$, $p > 0$ of tokens it produces is called the *production rate*.

As an example, consider the cleaning and boxing of eggs. The cleaning machine takes four eggs at a time and a packing machine packs ten eggs at a time in a box. The inputs are eggs and the outputs are boxes of ten eggs. This can be modelled with the following multi-rate dataflow graph.



Firings of the **Clean** actor require four eggs from the input to be enabled and produce four clean eggs every time they complete. Firings of the actor **Box** require ten eggs to be produced by **Clean** to be enabled. In figures, we write the production rates as a number at the producing end of a channel and the consumption rates at the consuming end of a channel.

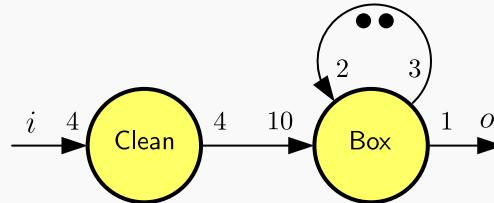
In single-rate dataflow graphs dependencies between actors are one-to-one. In multi-rate dataflow graphs they can be one-to-one, many-to-one, one-to-many or many-to-many, depending on the production and consumption rates. However, because the production and consumption rates are constant, it is possible in a well-behaved multi-

rate dataflow model to find a number of firings for each of the actors, i.e., a mapping $\rho : A \rightarrow \mathbb{N}$, such that $\rho(a) > 0$ for all $a \in A$, and for every channel from actor a with production rate p to actor b with consumption rate c in the graph, the following equation, called the *balance equation* holds.

$$\rho(a) \cdot p = \rho(b) \cdot c$$

The equation states that with a collection of firings according to the numbers in ρ for every channel, the production of tokens onto the channel and the consumption of tokens from the channel are balanced, i.e., equal. For the example of the eggs, such a solution exists, $\rho(\text{Clean}) = 5$ and $\rho(\text{Box}) = 2$. The unique, smallest solution ρ is called the *repetition vector* of the multi-rate dataflow graph. $\rho(\text{Clean}) = 25$ and $\rho(\text{Box}) = 10$ is also a solution to the balance equations, but it has more firings than necessary to be balanced. A collection of firings according to the repetition vector is together called an *iteration* of the graph.

A multi-rate graph that does not have a repetition vector cannot have a balanced production and consumption of tokens and is not well-behaved. It is called *inconsistent*. The graph below, for example, is inconsistent. For any $\rho(\text{Box}) > 0$, $\rho(\text{Box}) \cdot 3 \neq \rho(\text{Box}) \cdot 2$. In terms of the operational interpretation, more and more tokens would accumulate on the channel over time, which does not represent meaningful behavior.

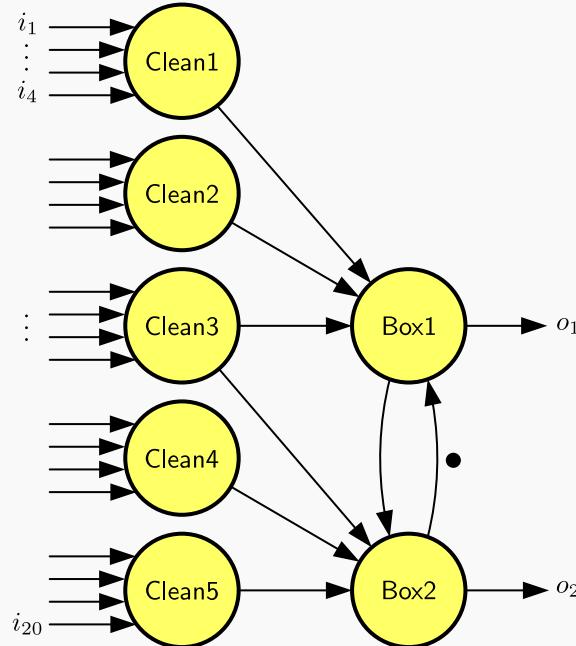


If a repetition vector does exist, the graph is called *consistent*.

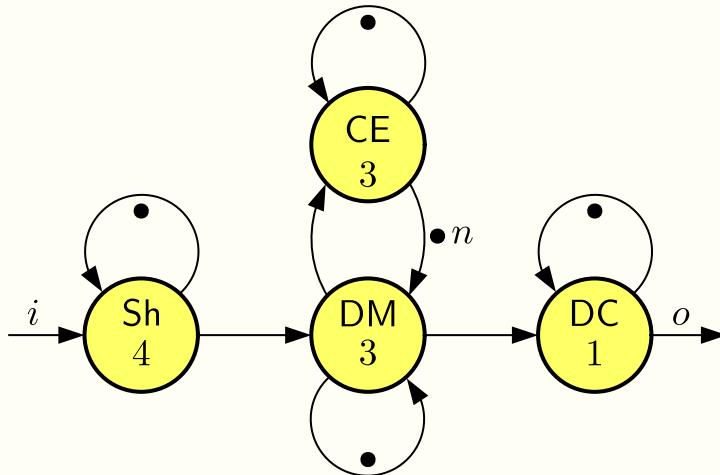
We will not present a precise mathematical definition of the constraints of a multi-rate dataflow graph. Instead, we show how a consistent multi-rate dataflow graph with repetition vector ρ can be converted to a single-rate dataflow graph.

The conversion proceeds by creating $\rho(a)$ single-rate actors for every actor a in the multi-rate graph. The graph below shows the single-rate version of the graph above. There are $\rho(\text{Clean}) = 5$ ‘copies’ of the actor **Clean** and $\rho(\text{Box}) = 2$ actors for the actor **Box**. The first firing of the original actor **Clean** will be represented by the first firing of the new actor **Clean1**, the second firing of **Clean** will be represented by the first firing of the new actor **Clean2**, and so on. The dependencies between firings are reconstructed by new channels, for example, the firing of **Box1** needs ten eggs, which are produced (4+4+2) by the first three firings of **Clean** and thus it depends on **Clean1**, **Clean2** and **Clean3**. Similarly, **Box2** depends on **Clean3**, **Clean4** and **Clean5**. The sixth firing of **Clean** can then be represented by the second firing of **Clean1**, because ρ satisfies the balance equations, and so on. Note how the self-edge of the **Box** actor has been converted to a loop between **Box1** and **Box2**. The original channel represents a dependency of firing k of **Box** on firing $k - 1$. After the conversion firing k and firing $k - 1$ are represented by different actors. Had the original channel two initial tokens, the dependency would be between firing k and firing $k - 2$, which are represented by the same actor after

conversion. In that case, actors Box1 and Box2 would both have had a self-edge with one initial token. Further, to comply with the single-rate inputs, we now need to distribute (in the model, not on the farm) the 20 eggs among 20 inputs, and similarly the output boxes are split.



Exercise C.1 (A wireless channel decoder – understanding dataflow). Consider the following dataflow graph, which provides a simple model of a Wireless Channel Decoder (WCD).



The WCD has an input i over which it receives modulation symbols and an output o through which it outputs decoded data. The Shift actor Sh maintains synchronization with the transmitter, the DM actor DeModulates the incoming signal, the CE actor Estimates the radio Channel, and the DC actor DeCodes the symbol. Channel estimation and decoding may be done in parallel. The number of tokens on the channel

from the channel-estimation actor **CE** to the demodulation actor **DM** determines how old the channel-estimation information is that **DM** uses. If $n = 1$, then **DM** uses information based on the previously received symbol. If $n = 2$, it uses information that is two symbols old.

1. Create a Gantt chart of the execution of the WCD model with $n = 1$ for six inputs that arrive at times 0, 4, 8, 12, 16, and 20, respectively. Assume that all actor firings occur as soon as possible. (We suggest to create the Gantt chart manually first and then check your answer with the CMWB.)
2. Create a Gantt chart of the execution of the WCD model with $n = 2$ for the same six inputs.
3. What is the maximal decoding rate of the decoder? What is the most recent channel-estimation information that can be used for demodulation at the maximally possible decoding rate?

See answer

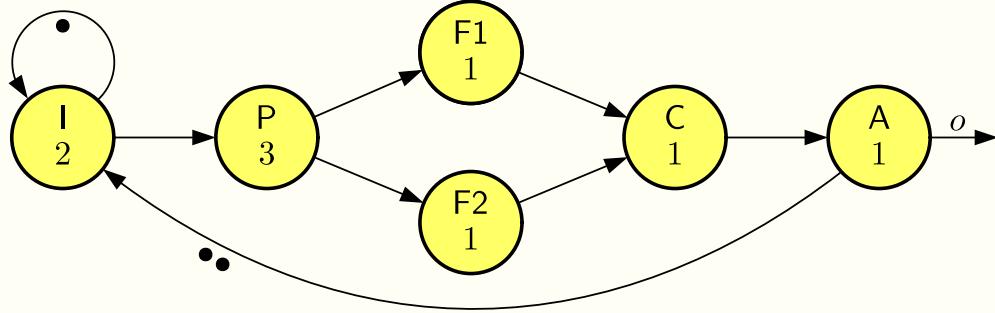
Exercise C.2 (A producer-consumer pipeline – creating dataflow models). Consider a simple producer-consumer pipeline consisting of a Producer actor **P**, a Filter actor **F**, and a Consumer actor **C**. Any produced sample is first filtered and then consumed again. Assume that the producer actor has an execution time (firing duration) of 2 time units to produce one sample and that the consumer actor has an execution time of 1 time unit per sample. The execution time of **F** varies based on the function being applied. With function f_1 it has an execution time of 1; with f_2 , it has an execution time of 3. Samples are produced and consumed one at a time, but the filtering of multiple samples may occur in parallel. The producer and the consumer are coupled to the filter with two buffers, each having space for two samples. Initially, the two buffers are empty. An actor claims space for producing an output sample into a buffer at the start of firing; it releases space of a consumed sample at the completion of firing.

1. Create a dataflow model that models this producer-consumer pipeline.
2. Provide a Gantt chart that shows the processing of the first six samples when applying function f_1 .
3. Provide a Gantt chart of the processing of the first six samples with function f_2 .

See answer

Exercise C.3 (An image-based control system – understanding and creating dataflow models). Consider the following dataflow graph modeling a simple Image-

Based Control (IBC) pipeline.



The pipeline has two actors, I and P , that capture an Image and Process it to extract two features. These two Features are processed by actors $F1$ and $F2$. Based on the features the pipeline computes a Control action in Actor C and eventually actuates the controlled plant with Actor A through output o . The model illustrates IBC pipelines as are used in, for example, autonomous vehicles. Note that the I actor with a channel to itself (a self-loop) with one initial token models a periodic image source with period 2.

1. Create a Gantt chart for the first six iterations, executing as soon as possible. (We suggest to create the Gantt chart manually first and then check your answer with the CMWB.)

We define the *average actuation rate* as the average number of outputs over time, ignoring any initial start up behavior in which the number of actuation actions may be lower. We define the *actuation delay* as the time between the start of an I firing and the corresponding o actuation output.

2. What is the rate at which the IBC pipeline actuates? What is the actuation delay?
3. The number of tokens on the channel from A to I is called the pipelining depth. What is the minimal pipelining depth at which the pipeline operates at maximum output rate? What is the maximum output rate?

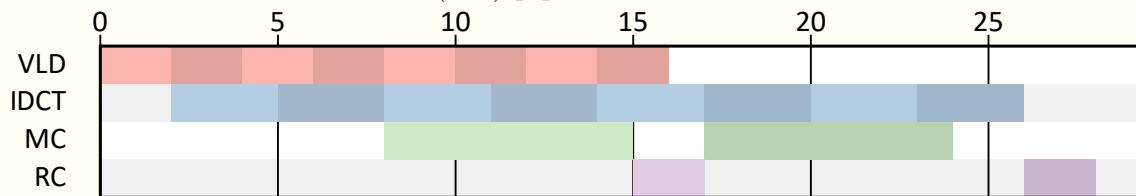
The next step is to implement the (optimized) IBC pipeline. Assume actors I , P , C , and A are each mapped onto their own processor that executes an actor firing with the given execution time. Assume actors $F1$ and $F2$ are scheduled together on a processor, where firings take one time unit and are scheduled alternatingly, starting with an $F1$ firing. The implemented IBC pipeline is referred to as the IBC System (IBCS). The IBCS should operate at the maximum output rate derived earlier.

4. Adapt the IBC model so that it properly captures the IBCS.
5. Create a Gantt chart for the first six iterations of the IBCS.
6. The IBCS does not operate at the required rate. Which processor(s) should be sped up to achieve the maximum actuation rate? How much?

7. The resource cost of the IBCS can be reduced while preserving its actuation rate. How can this be done? Adapt the IBCS model to reflect the proposed implementation change and provide a Gantt chart for the first six iterations of the adapted model.
8. Actuation delay is important in control applications. What is the actuation delay of the IBCS of the previous item? Do you see opportunities to improve the actuation delay?

See answer

Exercise C.4 (A video decoder - multi-rate dataflow). Consider the following Gantt chart of a Video Decoder (VD) pipeline.



The Variable-Length Decoding actor VLD decodes macroblocks of an incoming video frame, the IDCT actor performs an Inverse Discrete Cosine Transform on a macroblock, the MC actor performs Motion Compensation, and the ReConstruct actor RC composes a frame from the decoded macroblocks.

1. Give a dataflow model that generates the given Gantt chart.
2. What is the repetition vector of the dataflow model?
3. Convert the dataflow model to a single-rate dataflow model.
4. Create a Gantt chart for the first two iterations of the single-rate model

See answer

Exercise C.5 (An image-based control system – multi-rate dataflow). Consider again the Image-Based Control System (IBCS) of Exercise C.3. We continue with the further development of the optimized system given in the answer to the last item of that exercise. Assume the processing actor P is adapted to an actor P^+ that extracts four extra features of a different type than the original two features. These new features can be processed twice as fast as the original features on a normal processor, by an actor FF (for Fast Features), where each firing processes one feature. Also the actor that computes the control action is updated, to C^+ , to include the information of the new features. So it uses both the four new features and the two earlier ones.

1. Adapt the dataflow model of the last item of Exercise C.3 to a multi-rate dataflow

model of the IBCS that processes the four new features by Actor FF on Processor P3 and the two earlier features by Actor F (just another instance of the F1 and F2 actors, processing one feature per firing) on Processor P4.

2. What is the repetition vector of this model?
3. Convert the model to a single-rate model.

(Hint: Check the correctness of your models by comparing the Gantt charts of the two models in the CMWB.)

We now create a variant of the IBCS in which the workload of the feature processing is distributed equally over processors P3 and P4. Each processor receives two fast features and one of the original features. Each of the processors hosts a copy of the FF and F actors. Each processor first processes the two fast features and then the slow feature.

4. Create a multi-rate dataflow model for this IBCS.
5. Give the repetition vector for this model.
6. Convert the model to a single-rate dataflow model.

See answer

C.2 Schedules and performance metrics

Dataflow graphs capture the dependencies between operations in a system and they model how much time the execution of the operations takes. We have seen behaviors of such models, where the actor firings occur as soon as all of their dependency constraints are met. This is, in general, not always the case. The execution may postpone the start of an operation (the firing of the actor in the model) for various reasons, for example, because of limited resource to perform the operations. Dependencies may, however, never be violated. An operation may not occur before all its dependencies are satisfied. In this section, we consider the valid ways that a dataflow graph may be executed and we introduce related performance metrics.

Definition C.2 (Schedule). A schedule for a dataflow graph is a (possibly partial) function $\sigma : A \times \mathbb{N} \hookrightarrow \mathbb{R}$ that assigns starting times to the firings of actors of A , such that $\sigma(a, k)$ is the starting time of firing number k of actor a . We consider a number of properties that a schedule may have. - The schedule σ is called *valid* for given input arrivals $\alpha : I \times \mathbb{N} \hookrightarrow \mathbb{R}$ if and only if it satisfies the channel and input constraints:

- for all $(a_1, n, a_2) \in C$ and all $k \in \mathbb{N}$ such that $\sigma(a_2, k) = t$, either $k < n$ or $t \geq \sigma(a_1, k-n) + e(a_1)$.
- for all $(i, a) \in d_I$ and all $k \in \mathbb{N}$ such that $\sigma(a, k) = t$, $t \geq \alpha(i, k)$. Unless stated

otherwise we tacitly assume schedules to be valid, since invalid schedules are generally useless.

- σ is called a *self-timed* schedule, or *as-soon-as-possible* (asap) schedule, if all actor firings in σ start as soon as possible. In models without inputs, or in which actor firings do not depend on inputs, there may not be an earliest firing time. In those cases we may explicitly add an earliest firing time, usually 0 for such firings.
- The output production of the schedule σ is the function $\pi : O \times \mathbb{N} \hookrightarrow \mathbb{R}$ such that $\pi(o, k) = \sigma(d_O(o), k) + e(d_O(o))$.
- σ is called an *as-late-as-possible* (alap) schedule for outputs $\pi : O \times \mathbb{N} \hookrightarrow \mathbb{R}$, if all actor firings in σ fire only if needed to produce the outputs in π and do so as late as possible.
- σ is called a *complete* schedule if all actors firings that are possible are assigned a starting time in σ .
- σ is called *periodic*, with *period* μ , if for all $a \in A$, $k > 0$, such that $\sigma(a, k) = t$, then $\sigma(a, k - 1) = t - \mu$. If the dataflow graph is a multi-rate graph with repetition vector ρ , σ is called *periodic*, with *period* μ , if for all $a \in A$, $k \geq \rho(a)$, such that $\sigma(a, k) = t$, then $\sigma(a, k - \rho(a)) = t - \mu$.
- σ is called *finite* if $\sigma(a, k)$ is defined for only finitely many (a, k) . It is called *infinite* otherwise.
- σ is called *live* if $\sigma(a, k)$ is defined for infinitely many k for all actors $a \in A$.

Example C.3 (Schedule). The schedule that agrees with the Gantt chart of Figure C.2 is the following.

$$\sigma(A, 0) = 1, \sigma(A, 1) = 4, \sigma(A, 2) = 7, \sigma(A, 3) = 10, \sigma(B, 0) = 2, \sigma(B, 1) = 5, \\ \sigma(B, 2) = 8, \sigma(B, 3) = 11, \sigma(C, 0) = 2, \sigma(C, 1) = 5, \sigma(C, 2) = 8, \sigma(C, 3) = 11.$$

Schedule σ is, of course, valid, for the given input. It is also self-timed, and complete, since additional inputs are required to enable further actor firings. σ is also periodic, with period 3 and it is finite.

An alap schedule for the output productions $\pi(o, 0) = 10, \pi(o, 1) = 10$ is the following schedule. $\sigma_l(A, 0) = 2, \sigma_l(A, 1) = 5, \sigma_l(B, 0) = 3, \sigma_l(C, 0) = 6, \sigma_l(C, 1) = 6$

It is valid for the inputs of Figure C.2. Note that B only needs to fire once.

A live schedule may not always exist even if infinite input is offered to the dataflow graph. And for a finite number of inputs, a schedule may not exist that consumes all input, i.e., in which all actor firings that depend on that input are assigned a starting time. Graphs for which this applies have deadlocks.

Definition C.3 (Deadlock). A dataflow graph is said to *deadlock* if there exist $a \in A$ and $k \in \mathbb{N}$ such that firing k of actor a directly or indirectly has a dependency on its own completion.

Theorem C.1. A single-rate dataflow graph deadlocks if and only if the dataflow graph has a cycle of channel dependencies without initial tokens on them.

A deadlocking single-rate dataflow graph can easily be visually spotted by checking for cycles without initial tokens. For multi-rate dataflow graphs it is less easy. Because it produces and consumes larger quantities of tokens, there may be initial tokens on every cycle, but still not enough for the graph to be free of deadlock. It can be shown however, that a multi-rate dataflow graph is deadlock free if and only if there exists a schedule that assigns start times to all firings in a single iteration of the graph. This can be effectively algorithmically checked.

We take a particular interest in the self-timed schedule of dataflow graphs. We use the property that there cannot be different, but both self-timed schedules.

Theorem C.2. If a dataflow graph has a self-timed (asap) schedule for input arrivals $\alpha : I \times \mathbb{N} \hookrightarrow \mathbb{R}$, then it is unique.

Note that the property of Theorem C.2 holds true, whether or not we add additional constraints such as an earliest firing time of 0.

With schedules of a dataflow graph we can associate *performance metrics*. We introduce three metrics in particular. *Throughput* intuitively refers to the amount of processing that a system can perform, on average, in the long run, per unit of time. It is relevant for indefinite operation. *Makespan*, in contrast, is a performance metric that can be associated with finite behaviors. It refers to the total amount of time that it takes to complete a job. Finally, *latency* is a metric that characterizes the time span between the start and the completion of the processing of individual elements. Since this time may differ for different inputs and outputs, latency refers to the maximum difference. Precise definitions are given below.

Definition C.4 (Throughput). Given a dataflow graph, an actor $a \in A$ and a schedule σ such that $\sigma(a, k)$ is defined for all $k \in \mathbb{N}$, the *throughput* of actor a in schedule σ is given by the following limit, if it exists:

$$\tau(\sigma, a) = \lim_{k \rightarrow \infty} \frac{k}{\sigma(a, k)}$$

Otherwise, the throughput of a in σ is not defined.

If σ is a periodic schedule with period μ , then the throughput of any actor, if it fires infinitely often, is equal to $1/\mu$.

Definition C.5 (Makespan). Given a dataflow graph with a finite schedule σ . The *makespan* of σ is

$$M(\sigma) = \max_{(a,k) \in \text{dom}(\sigma)} \sigma(a, k) + e(a)$$

Definition C.6 (Latency). Given a dataflow graph, input arrivals $\alpha : I \times \mathbb{N} \hookrightarrow \mathbb{R}$, a selected input $i \in I$ and a selected output $o \in O$ with productions π under σ , let K be the set of indices for which input on i and output on o are defined:

$$K = \{k \in \mathbb{N} \mid (i, k) \in \text{dom}(\alpha) \text{ and } (o, k) \in \text{dom}(\pi)\}$$

then *latency* is defined as

$$L(\sigma, i, o) = \sup_{k \in K} \pi(o, k) - \alpha(i, k)$$

In the definition of latency, \sup refers to the *supremum*, or least upper bound. It is similar to the maximum, but takes care of the situation that there may not exist a unique k for which the value is the maximum when there are infinitely many $k \in \text{dom}(\alpha)$.

Performance metrics are associated with the schedules of the graph. We are often interested in the best possible schedule for a given metric. We can then also associate a performance metric directly with the graph, by which we mean the performance of the best possible schedule. Usually, this is the self-timed schedule. The maximum throughput of a dataflow graph, for example, is the throughput of the self-timed schedule of the graph.

Example C.4 (Performance metrics). In the schedule of Example C.3, the performance metrics are as follows.

It does not have a throughput, because it is a finite schedule. However, if we briefly imagine an infinite periodic continuation of the given schedule, then the throughput of any actor in that schedule is $\frac{1}{3}$.

The makespan is equal to the time of the latest completion in the schedule, i.e., $M(\sigma) = \sigma(C, 3) + e(C) = 15$. The Gantt chart is shown in Figure C.2. Makespan can easily be determined from it as the last point in time where there is any activity in the chart. The latency is equal to the maximum distance between an input and the corresponding output. The largest distance occurs at the last pair of input-output. Therefore, $L(\sigma, i, o) = 15 - 6 = 9$.

Example C.5 (A Railroad Network). Dataflow is very suitable to model a railroad network. Activities have known durations and there are dependency constraints between operations. Figure C.6 shows a small example of a railroad network. The blue rectangles are railroad stations at different cities. We consider the stations Amsterdam, The Hague, Eindhoven and Maastricht to be part of the network we want to model. Trains may be arriving from Liège, which we will consider as inputs to our system and trains may be departing to Schiphol, which we consider as outputs of our system. The railroad trajectories between the stations have known nominal traveling durations as shown in the figure.

Operation of the railroad network needs to comply with certain dependencies. New trains can only depart from a station when all trains that come from neighboring stations have arrived, so as to give the passengers the opportunity to change trains.

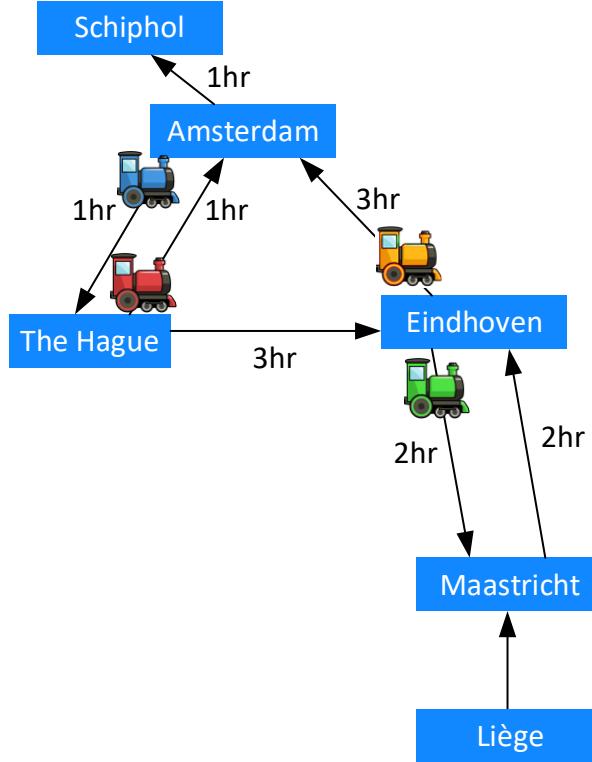


Figure C.6: A railroad network

For simplicity we assume that changing trains does not take any time. To avoid a deadlock, some trains need to initially depart without any trains coming in. We assume there are four trains as shown in the figure.

A dataflow graph of the railroad network is shown in Figure C.7. The blue actors correspond to the stations. AMS for Amsterdam, THG for The Hague, EHV for Eindhoven and MST for Maastricht. Their input and output dependencies model the synchronization of trains waiting for connection. Their firing durations are equal to 0 as we assume that changing trains does not take time. The red colored actors represent the railroad tracks. their firings start when a train departs from the station it consumes from and complete when the train arrives at the next station. Note how the four initial tokens in the graph represent the four trains that initially depart.

Since some of the firings in the dataflow graph do not depend on any inputs, we assume that their earliest starting time in a self-timed execution is equal to 0.

Figure C.8 shows a Gantt chart of the self-timed execution of the dataflow graph of the railroad network. After an initial startup phase it assumes a repetitive pattern. The makespan of the schedule is 24hrs. The latency from Liège to Schiphol is 4hrs.

Exercise C.6 (A wireless channel decoder – schedules, performance). Consider again the Wireless Channel Decoder (WCD) model of Exercise C.1. Consider both the options with $n = 1$ and $n = 2$.

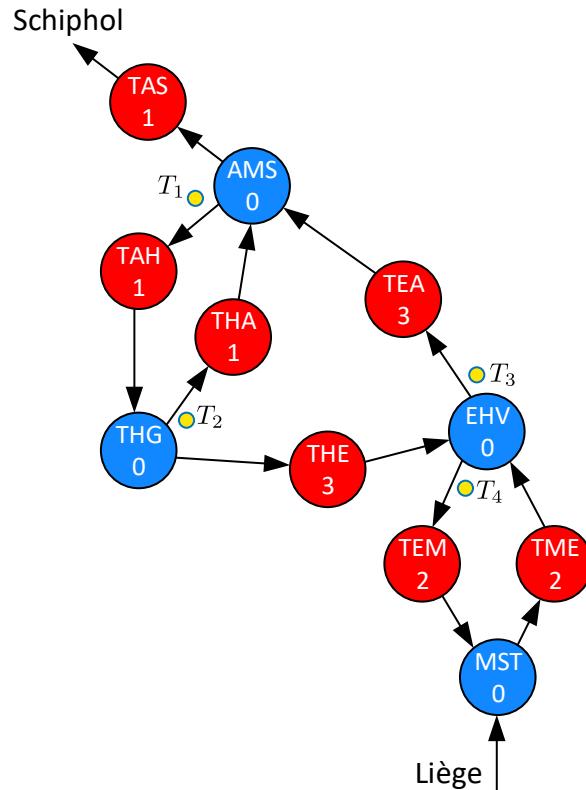


Figure C.7: Dataflow model of the railroad network

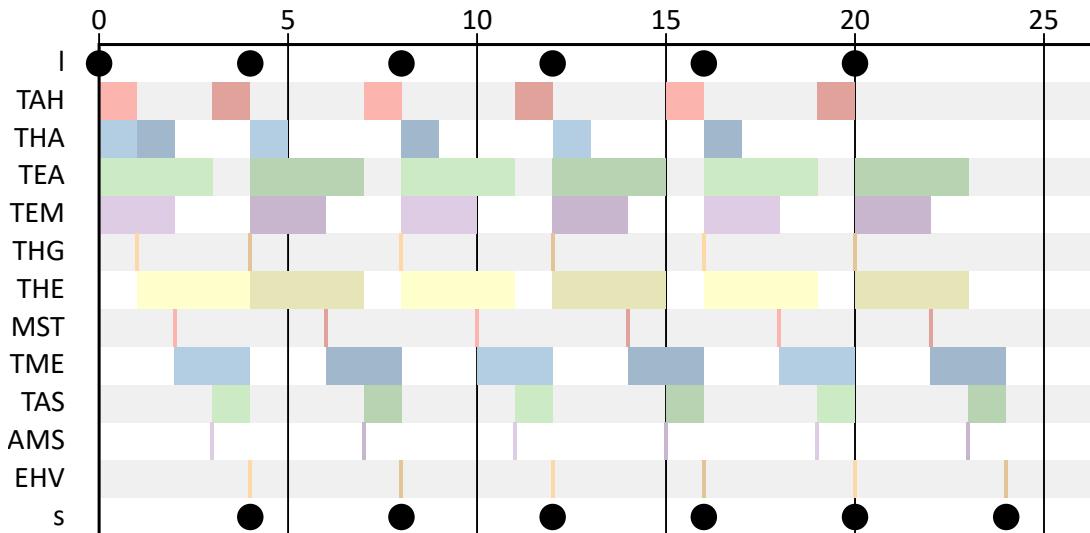


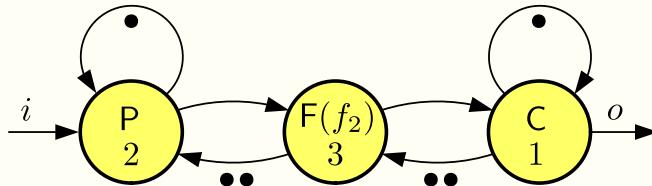
Figure C.8: Gantt chart of the railroad network

1. What is the makespan of the self-timed schedules for six iterations?
2. What is the latency between inputs on i and productions on o in those schedules?

3. Are the schedules periodic?
4. Provide alap schedules for the two cases. Are those alap schedules periodic?
5. Consider an infinite extension of the arrivals with a period of 4. Consider the infinite self-timed scheduled processing those inputs for both $n = 1$ and $n = 2$. What is the throughput of the decoder actor DC in these schedules?

See answer

Exercise C.7 (A producer-consumer pipeline – schedules, performance). Consider the following variant with explicit inputs and outputs of the producer-consumer pipeline of Exercise C.2, with Filter actor F instantiated with filtering function f_2 .



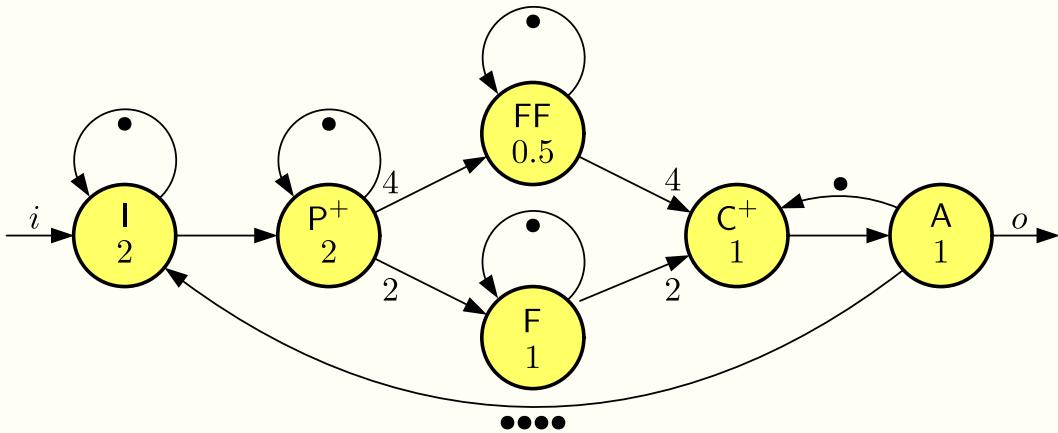
Assume inputs arrive periodically with period 2 and the first input arrives at time 0.

1. What is the makespan of the self-timed schedule for six iterations?
2. What is the latency between inputs on i and productions on o in that schedule?
3. Is the schedule periodic?
4. Provide an alap schedule for the output production of the self-timed schedule.
5. What is the throughput of Actor C in the infinite self-timed schedule?
6. Recall that the dataflow graph models buffer capacities of 2 between the filter and the producer and consumer. What is the maximal achievable throughput for the producer-consumer pipeline (Actor C) if the buffer capacities may be enlarged? What are the minimal buffer sizes that realize this throughput?
7. Is the self-timed schedule that maximizes throughput for the minimal buffer capacities of the previous item periodic?
8. Create an alap schedule for the maximal-throughput productions.

See answer

Exercise C.8 (An image-based control system – schedules, performance).

Consider the following variant of the Image-Based Control System (IBCS) of Exercise C.5.



Assume inputs arrive periodically with period 2 and the first input arrives at time 0.

1. What is the makespan of the self-timed schedule for six iterations?
2. What is the $i\text{-}o$ latency in that schedule?
3. Is the schedule periodic?
4. Provide an alap schedule for the output production of the self-timed schedule of the first six iterations.
5. What is the throughput of Actor A in the infinite self-timed schedule?

See answer

C.3 Max-Plus Algebra

In the previous sections we have introduced timed dataflow models and their schedules and performance metrics to study timed deterministic discrete-event systems. To develop methods for performance analysis, in this module, we introduce an algebra which will help us to perform the required analysis.

To track the evolution of a system in time we label events (for example, the start of a firing, or operations in a schedule) with *time stamps*, real-valued numbers that indicate at what point in time an event occurs. Events can happen at any (real-valued) point in time. They can also have negative time stamps.

Definition C.7 (Time Stamp). A *time stamp* is a value from the set $\mathbb{R} \cup \{-\infty\}$. We denote the set of time stamps as $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{R} \cup \{-\infty\}$.

Note that in the mathematical definition of a time stamp we also allow a time stamp to have the peculiar special value $-\infty$. (Note that $-\infty \notin \mathbb{R}$ and neither is ∞ , which is also not in \mathbb{T} .) We will see later that this is a very convenient mathematical abstraction. In the

real world events do not occur at time $-\infty$! But $-\infty$ is earlier than any ‘real’ time stamp.

Example C.6 (Dataflow Equations). Consider a single-rate dataflow actor a having a firing duration $d = e(a)$, with a single input channel i and a single output channel o . Assume that we know the time stamp x_i of the arrival-event of the input token to channel i . Then, we can easily compute the time stamp x_o of the production-event of the output token produced on o when the actor a completes its self-timed firing:

$$x_o = x_i + d$$

Now, assume that the actor a has two input channels, i_1 and i_2 . A firing of a consumes one token from each of its inputs. We can again compute the time stamp of the production of the output token from the time stamps of the input tokens if a fires in a self-timed manner. Recall that the self-timed execution means that the actor fires as soon as both tokens are present. If we consider the start of the firing of a as an event with time stamp x_a , then

$$x_a = \max(x_{i_1}, x_{i_2})$$

and

$$x_o = x_a + d = \max(x_{i_1}, x_{i_2}) + d = \max(x_{i_1} + d, x_{i_2} + d)$$

The very simple Example C.6 illustrates that the behavior of a timed dataflow graph can be described in terms of the time stamps of the events that occur in the model and that the two important mathematical operations that describe this behavior are the max operation and the $+$ operation.

The operations work on time stamps from \mathbb{T} as defined in Definition C.7. A mathematical definition of the *algebra* of the set of time stamps with the addition and maximum operations is given in Definition C.8. We introduce two new symbols for the operations, because they are commonly used in literature and they are more convenient to write than the conventional notation, especially for max. Observe also that we must then define how the operators work on the special time stamp $-\infty$, for which the common addition and maximum operations are not defined. The way they are defined to work on $-\infty$ is however in line with our ‘intuition’ about $-\infty$.

Definition C.8 (Max-Plus Algebra). Max-plus algebra is the algebra $(\mathbb{T}, \oplus, \otimes)$. It works on the set \mathbb{T} of time stamps and includes the operators \oplus and \otimes , which are defined as follows.

$$x \oplus y = \begin{cases} x & \text{if } y = -\infty \\ y & \text{if } x = -\infty \\ \max(x, y) & \text{if } x, y \in \mathbb{R} \end{cases}$$

$$x \otimes y = \begin{cases} -\infty & \text{if } x = -\infty \text{ or } y = -\infty \\ x + y & \text{if } x, y \in \mathbb{R} \end{cases}$$

The operators of max-plus algebra have the following properties.

- *Associativity:* $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ and $x \otimes (y \otimes z) = (x \otimes y) \otimes z$;

- *Commutativity:* $x \oplus y = y \oplus x$ and $x \otimes y = y \otimes x$;
- *Distributivity:* $x \otimes (y \oplus z) = x \otimes y \oplus x \otimes z$;
- *Zero element:* $-\infty$ is a zero element of the \otimes operator, $-\infty \otimes x = -\infty$ for all x ; \oplus has no zero element;
- *Unit element:* $-\infty$ is a unit element of the \oplus operator, $-\infty \oplus x = x$ for all x ; 0 is a unit element of the \otimes operator, $0 \otimes x = x$ for all x ;
- \otimes takes precedence over \oplus , so that $x \otimes y \oplus z$ means $(x \otimes y) \oplus z$.

The symbols, \otimes and \oplus for the addition and max operators are not selected arbitrarily. We can observe that the algebraic properties of max-plus algebra as introduced above are almost identical to the properties of the algebra of real numbers with the operators \times and $+$ that we are all very familiar with. In the analogy, the following correspondences hold

- The \otimes operator corresponds to the \times operator (hence the symbol);
- The \oplus operator corresponds to the $+$ operator (hence the symbol);
- The element $-\infty \in \mathbb{T}$ corresponds to the element $0 \in \mathbb{R}$;
- The element $0 \in \mathbb{T}$ corresponds to the element $1 \in \mathbb{R}$;

The analogy helps us to get acquainted with the new algebra using familiarity with the traditional algebra. Check that the algebraic rules and properties mentioned above for max-plus algebra all remain valid under this correspondence.

Example C.7 (Conveyor Belt). A conveyor belt (see Figure C.9) is used to transport objects from one site to another, as quickly as possible. The length of the belt is l and it moves with a velocity v . The objects that are transported have a size of d . Three objects arrive at the time stamps: o_1, o_2, o_3 . The order of arrival is unknown, but the objects need to be transported in order (1, 2, 3). We derive max-plus-algebraic equations for the time stamps of the arrival events, a_1, a_2 and a_3 , of the three objects at the end of the belt. The first object is independent of the others, arrives at time o_1 , takes l/v time units for transportation, so it arrives at

$$a_1 = o_1 + \frac{l}{v} = \frac{l}{v} \otimes o_1.$$

Note that we think of $\frac{l}{v}$ as a constant and o_1 as a variable, and therefore it is more natural to write $\frac{l}{v} \otimes o_1$ than $o_1 \otimes \frac{l}{v}$, although they are identical, just like in regular algebra we prefer to write $3 \cdot x$ over $x \cdot 3$.

The second object cannot occupy the same space on the belt and needs to be transported second. The belt is clear for the second object $\frac{d}{v}$ time units after the arrival of object 1. So, the second object is put on the belt when the belt is clear, or, when the second object arrives, whichever is later. We can derive that the arrival at the end

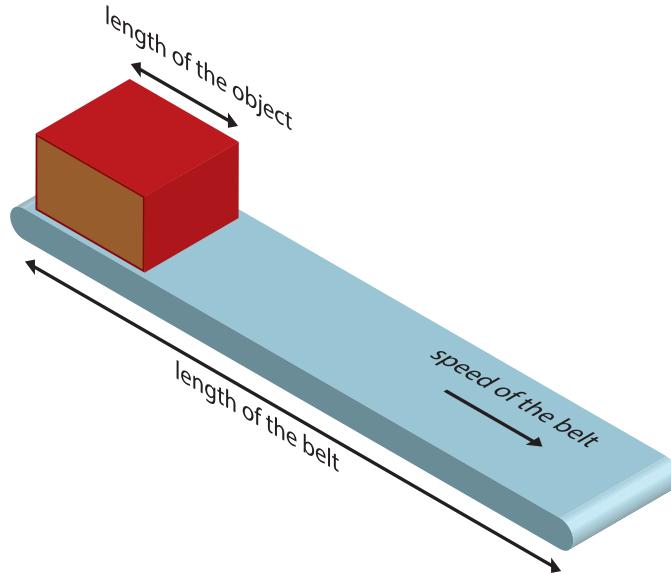


Figure C.9: Conveyor belt.

of the belt occurs at time

$$a_2 = \max(o_1 + \frac{d}{v}, o_2) + \frac{l}{v} = \frac{l}{v} \otimes (\frac{d}{v} \otimes o_1 \oplus o_2) = \frac{l+d}{v} \otimes o_1 \oplus \frac{l}{v} \otimes o_2.$$

With similar reasoning, the third object can be shown to arrive at time

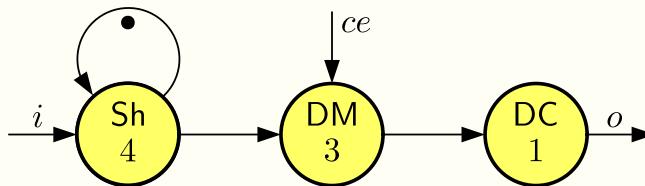
$$a_3 = \frac{l+2d}{v} \otimes o_1 \oplus \frac{l+d}{v} \otimes o_2 \oplus \frac{l}{v} \otimes o_3$$

It is then also not difficult to generalize the result for more than three objects to object n arriving at time

$$a_n = \bigoplus_{1 \leq k \leq n} \frac{l + (n-k)d}{v} \otimes o_k$$

Note that we use the \bigoplus notation to represent repeated application of the \oplus operator in the same way that the familiar \sum notation is used for repeated application of the $+$ operator.

Exercise C.9 (A wireless channel decoder – max-plus analysis). Consider the following fragment of the Wireless Channel Decoder (WCD) dataflow model of Exercise C.1.



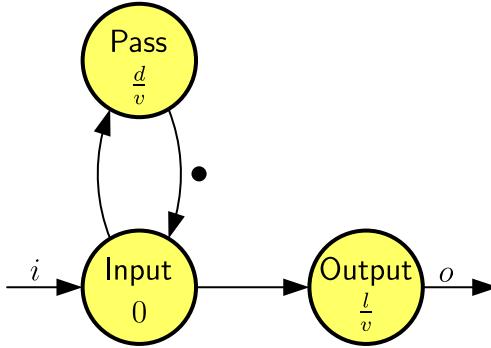


Figure C.10: Dataflow model of the conveyor belt.

We want to analyze how the output production times depend on the input arrivals and the availability of channel-estimation information. The WCD fragment is a dataflow graph with two inputs, the symbol input i and the channel-estimation input ce . Executing the dataflow model gives a sequence of token production times on symbol output o , referred to as $x_o(0), x_o(1), \dots$. Those token production times depend on the arrival times of tokens on the inputs i and ce , denoted $x_i(0), x_i(1), \dots$ and $x_{ce}(1), x_{ce}(2), \dots$, respectively. Assume that the initial token is available at time 0 and that inputs do not have negative arrival times.

1. Give expressions in max-plus notations and in the usual notations of real-number algebra for $x_o(0)$ in terms of $x_i(0)$ and $x_{ce}(0)$.
2. Give expressions in max-plus notations and in the usual notations for $x_o(1)$ in terms of $x_i(0), x_i(1), x_{ce}(0), x_{ce}(1)$.
3. Provide a condition on the arrival times of inputs on i and ce so that production times $x_o(k)$ do not depend on the arrival times $x_{ce}(k)$.

See answer

C.4 Max-Plus Linear Systems

C.4.1 Event Sequences

We often consider *sequences* of events. For instance, a sequence of (arrivals of) products that need to be manufactured, or a sequence of data items that need to be processed. Such sequences have a beginning, but do not need to have an end, i.e., they can be of *finite* length $n \in \mathbb{N}$, or they can be of infinite length. For performance analysis purposes, mathematically, we refer to sequences of time stamps of events. We call them, in short, *event sequences*. We denote sequences of events by lower case letters and we use parentheses to refer to the events in a sequence. $s(k)$ with $k \geq 0$ denotes event number k in event sequence s . If s is finite and has length n , then $s(k)$ is not defined for $k \geq n$. We refer to k as the *event index*.



Figure C.11: Conveyor belt model.

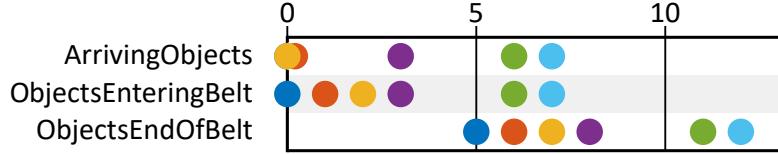


Figure C.12: Conveyor belt event sequences.

Definition C.9 (Event Sequence). An *event sequence* s is a function $s : \mathbb{I} \rightarrow \mathbb{T}$, where $\mathbb{I} = \{0, \dots, n-1\}$ for a finite sequence of length n and $\mathbb{I} = \mathbb{N}$ for an infinite sequence.

We illustrate event sequences, as shown in Figure C.12, by a time line with a row of dots at the time stamps of the events. Event indices are indicated by colors of the dots. Note that events in an event sequence do not necessarily have non-decreasing time stamps, i.e., it may be the case that $s(k) > s(k+1)$. Where needed, we use the following color coding to represent the event index, i.e., event with index 0 is blue, 1 is orange, 2 is yellow, 3 purple, etcetera.

We define the following operations on event sequences, because we need them later.

Definition C.10 (Delayed Event Sequence). If s is an event sequence, then s^N is the event sequence such that $s^N(k) = -\infty$ if $k < N$ and $s^N(k) = s(k-N)$ if $k \geq N$. If s is of finite length n then s^N is a finite event sequence of length $n+N$. If s is infinite, then s^N is infinite.

Definition C.11 (Maximum of Event Sequences). Let s_1 and s_2 be two event sequences of the same length. Then $s_1 \oplus s_2$ is an event sequence, also of the same length, such that $(s_1 \oplus s_2)(k) = s_1(k) \oplus s_2(k)$.

Definition C.12 (Scaled Event Sequence). Let s be an event sequence and $c \in \mathbb{T}$. Then $c \otimes s$ is an event sequence, of the same length, such that $(c \otimes s)(k) = c \otimes s(k)$.

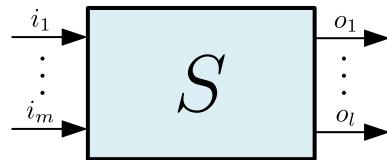
Definition C.13 (Zero Event Sequence). We use ϵ to denote the infinite event sequence with $\epsilon(k) = -\infty$ for all $k \in \mathbb{N}$.

Definition C.14 (μ -Periodic Event Sequence). The μ -periodic event sequence is the infinite event sequence s with $s(k) = \mu \cdot k$ for all $k \in \mathbb{N}$.

Figure C.13 shows an event sequence s and the event sequence s^3 which is delayed by 3



Figure C.13: Delayed event sequence

Figure C.14: A max-plus-linear system S

samples. Note that the time stamps are the same, except for the added time stamps of $-\infty$, which are not visible on the time line. The indices have changed, as can be observed from the modified colors. For example, $s^3(4) = s(1) = 6$.

We use the following notation for event sequences:

$$s = [t_0, t_1, t_2, \dots]$$

In this case, s is the event sequence such that $s(0) = t_0$, $s(1) = t_1$, $s(2) = t_2$, etcetera.

We use event sequences to study the performance of systems. We therefore define their throughput as follows.

Definition C.15 (Event-Sequence Throughput). If s is an infinite event sequence, then the throughput of s is given by the following limit, if it exists:

$$\tau(s) = \lim_{k \rightarrow \infty} \frac{k}{s(k)}$$

The throughput of the μ -periodic event sequence, for example, is $1/\mu$.

C.4.2 Event Systems

We are interested in systems that operate on one or more sequences of events. A system takes a number of input event sequences and produces one or more output event sequences. We consider systems that are deterministic. The output sequences are uniquely determined by the input sequences. If a system S produces for input sequences i_1, \dots, i_m the output sequences o_1, \dots, o_l , we write

$$i_1, \dots, i_m \xrightarrow{S} o_1, \dots, o_l$$

A visualization is shown in Figure C.14.

Example C.8 (Conveyor Belt System Model). The conveyor belt from Example C.7 can be described as an event system if we assume that it has a (finite or infinite) input sequence of objects arriving that need to be transported in a given order along

the conveyor belt. It is visualized in Figure C.11 as an abstract box CB with input i and output o . The system has a single output, the arrival events of the objects at the end of the conveyor belt. The output sequence is equally long as the input sequence, all boxes arrive at the end of the belt. It is also deterministic, following the equations presented in Example C.7.

We consider a class of event systems that have special properties. We look at properties that are akin to the classical concept of *linear time-invariant systems* (LTI systems).

Definition C.16 (Max-Plus-Linear Systems). An event system S is called *max-plus-linear* if for all event sequences i_k, j_k, o_k, p_k , and any $c \in \mathbb{T}$, the following properties hold:

- (additivity) if $i_1, \dots, i_m \xrightarrow{S} o_1 \dots, o_l$ and $j_1, \dots, j_m \xrightarrow{S} p_1 \dots, p_l$ then $i_1 \oplus j_1, \dots, i_m \oplus j_m \xrightarrow{S} o_1 \oplus p_1 \dots, o_l \oplus p_l$
- (homogeneity) if $i_1, \dots, i_m \xrightarrow{S} o_1 \dots, o_l$ then $c \otimes i_1, \dots, c \otimes j_m \xrightarrow{S} c \otimes o_1 \dots, c \otimes o_l$

Definition C.17 (Index-Invariant Systems). An event system S is called *index invariant* if for all event sequences i_k, o_k , and any $N \in \mathbb{N}$, the following property holds:

- if $i_1, \dots, i_m \xrightarrow{S} o_1 \dots, o_l$, then $i_1^N, \dots, i_m^N \xrightarrow{S} o_1^N \dots, o_l^N$

Additivity of an event system means that if a set of input sequences corresponds to the point-wise maximum of two (or more) input sequences, then the corresponding output, similarly, corresponds to the point-wise maximum of the individual outputs. This allows us to break down complex input sequences into smaller parts and analyze them separately, both along the different input sequences to the system and to the individual events in those sequences.

Homogeneity means that the system is insensitive to absolute time of the various events; it is sensitive only to the relative time differences. If we shift all inputs backward or forward by a fixed amount of time, then also all outputs will shift by the same amount of time. This property allows us to perform some analysis for an arbitrary absolute point in time and translate the results to other points in time.

Similarly, index-invariance allows us to limit our analysis to a given event index and translate the analysis results straightforwardly to other event indices. (This is a good moment to reconsider Exercise C.9.)

Example C.9 (Our Conveyor Belt is Linear and Index Invariant). In Example C.7 we have concluded the following equation to describe the conveyor belt event system $o \xrightarrow{CB} a$. (Note that in Example C.7 we started counting objects from 1 and

the indices of event sequences start from 0.)

$$a(n) = \bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes o(k)$$

- (additivity) Let $o_1 \xrightarrow{CB} a_1$, $o_2 \xrightarrow{CB} a_2$ and $o_1 \oplus o_2 \xrightarrow{CB} a_3$. Then

$$\begin{aligned} a_3(n) &= \bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes (o_1 \oplus o_2)(k) \\ &= \bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes o_1(k) \oplus \frac{l + (n - 1 - k)d}{v} \otimes o_2(k) \\ &= \left(\bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes o_1(k) \right) \oplus \\ &\quad \left(\bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes o_2(k) \right) \\ &= a_1(n) \oplus a_2(n) \end{aligned}$$

Thus, $o_1 \oplus o_2 \xrightarrow{CB} a_1 \oplus a_2$

- (homogeneity) Let $o \xrightarrow{CB} a$, and $c \otimes o \xrightarrow{CB} a'$. Then

$$\begin{aligned} a'(n) &= \bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes (c \otimes o)(k) \\ &= \bigoplus_{0 \leq k < n} c \otimes \frac{l + (n - 1 - k)d}{v} \otimes o(k) \\ &= c \otimes \bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes o(k) \\ &= c \otimes a(n) \end{aligned}$$

Thus, $c \otimes o \xrightarrow{CB} c \otimes a$

- (index invariance) Let $o \xrightarrow{CB} a$ and $o^N \xrightarrow{CB} a'$. Then

$$\begin{aligned}
a'(n) &= \bigoplus_{0 \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes (o^N)(k) \\
&= \left(\bigoplus_{0 \leq k < N} \frac{l + (n - 1 - k)d}{v} \otimes -\infty \right) \oplus \\
&\quad \left(\bigoplus_{N \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes o(k - N) \right) \\
&= \left(\bigoplus_{N \leq k < n} \frac{l + (n - 1 - k)d}{v} \otimes o(k - N) \right) \\
&\quad \{ \text{substitute } m = k - N \} \\
&= \bigoplus_{0 \leq m < n - N} \frac{l + (n - 1 - (m + N))d}{v} \otimes o(m) \\
&= \bigoplus_{0 \leq m < n - N} \frac{l + ((n - N) - 1 - m)d}{v} \otimes o(m) \\
&= \begin{cases} -\infty & \text{if } n < N \\ a(n - N) & \text{if } n \geq N \end{cases} \\
&= a^N(n)
\end{aligned}$$

Thus, $o^N \xrightarrow{CB} a^N$

Dataflow graphs can be seen as event systems by considering the token sequences arriving on inputs and produced on outputs as event streams. Dataflow graphs are, in general, not necessarily linear systems though, since their behavior is only determined with a schedule. As we will see later, however, our primary interest is in self-timed schedules of dataflow graphs. Assuming the availability of initial tokens at the earliest possible point in time, $-\infty$, and hence no artificial lower bound, 0 or otherwise, on the firing times, the self-timed schedule is uniquely determined by the input sequences. Actor firings that do not depend on any input then start (and complete) at time $-\infty$. As a result, a self-timed dataflow graph is deterministic and dataflow graphs with self-timed execution are linear and index-invariant systems.

Theorem C.3. A single-rate dataflow graph under self-timed execution and with initial tokens available at $-\infty$ is a linear and index-invariant system.

Theorem C.3 appears to restrict the application of properties of index-invariant systems to dataflow graphs to which no additional constraints have been applied on the earliest

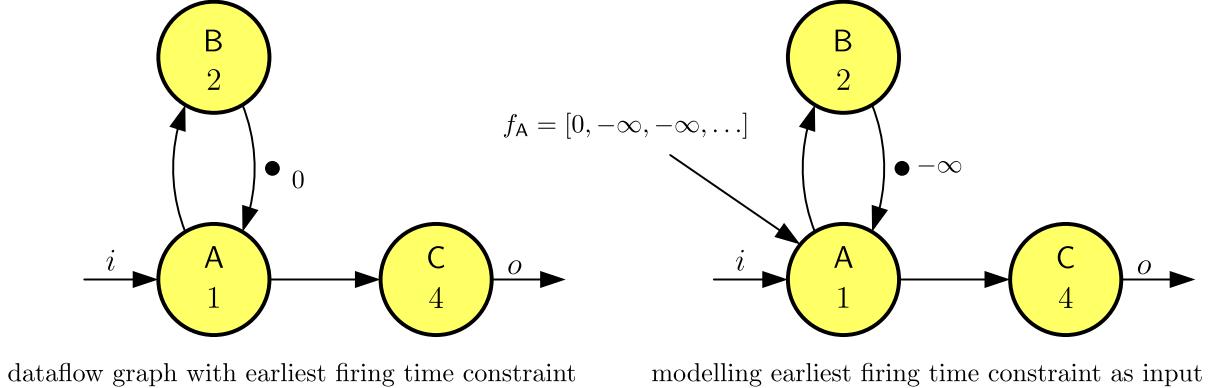


Figure C.15: Modelling earliest firing times in an event system

firing times, or to the availability of initial tokens. Indeed, such dataflow graphs need not be index-invariant. However, we can alternatively model the additional constraints by additional inputs, allowing the system to be modelled as an index-invariant system including the analysis opportunities this offers. Figure C.15 shows the dataflow graph of Figure C.1, with the applied constraint that actors cannot fire earlier than time stamp 0. This is indicated by the time stamp 0 label next to the initial token in the graph. This restricts actor A to not fire before time 0 and in this example that ensures that also other actors cannot fire before time 0. This behavior can also be modelled with the graph on the right hand side of the figure. This graph has no additional constraints on firing times, indicated by the time stamp of $-\infty$ on the initial token. There is however, an additional input added to actor A. The firing constraints can be enforced by applying the right input sequence to input f_A . The input sequence $f_A = [0, -\infty, -\infty, -\infty, \dots]$ enforces that no actor firing occurs before time 0, the equivalent of the graph on the left of the figure. Such an input can also be used to enforce other constraints. For example. $f_A = [0, 5, 10, 15, \dots]$ restricts A to (not fire earlier than) a periodic schedule. Using this approach to model additional firing constraints Theorem 2 applies to such dataflow graphs and analysis methods, introduced later, that depend on it can be applied.

Multi-rate dataflow graphs are also linear, i.e., satisfy both the additivity and homogeneity properties. They are however not index invariant. Since their actors consume and/or produce multiple tokens at a time, input events with different indices may encounter different actor firings leading to different, not just shifted, output sequences. In the multi-rate example of the eggs, for instance, the second egg ends up in the same output box as the first egg, not the second box as index invariance would require.

We have seen, however, that a multi-rate graph can be converted to a single-rate graph, which is index invariant. In case of the eggs, it has twenty (egg) inputs and two (box) outputs. This means that if we shift by twenty input events and two output events, the response is invariant.

In general, multi-rate dataflow graphs do obey a property that is similar to index invariance, but it requires the indices to be shifted in accordance with a multiple of

the repetition vector.

Exercise C.10 (A wireless channel decoder – max-plus-linear index-invariant system). Reconsider the fragment of the Wireless Channel Decoder (WCD) dataflow model of Exercise C.9. Show that this dataflow graph models a max-plus-linear index-invariant event system. For the purpose of this analysis, consider the arrival/production times on the inputs resp. output as event sequences. You may assume that the event sequence corresponding to the production times of tokens on the self-loop of Actor Sh is $(4 \otimes i)^1$, when i is the event sequence corresponding to the input channel i and $_1^1$ is the delay operator of Definition C.10. (The observant reader may notice that this assumption is only valid if inputs on channel i arrive with intermediate arrival times of at least 4. We come back to this aspect in later exercises.)

See answer

C.4.3 Superposition

If a system is linear then we can apply a divide-and-conquer strategy to analyze how it responds to complex stimuli. For example, the manufacturing system of Example C.2 is a max-plus-linear event system.

$$in_{top}, in_{bottom} \xrightarrow{MS} o$$

Additivity says that if $in_{top}, \epsilon \xrightarrow{MS} o_1$ and $\epsilon, in_{bottom} \xrightarrow{MS} o_2$, then $o = o_1 \oplus o_2$.

Assume that $in_{bottom} = [0, 6, 22, 24]$ and $in_{top} = [3, 9, 13, 12]$. Figure C.16 shows the response of the system to in_{top}, in_{bottom} , to in_{top}, ϵ and to ϵ, in_{bottom} . It is easy to verify that the output in the first case is indeed the maximum of the two output sequences of the second and the third case. In fact, observe that it is true for the entire Gantt chart that for any actor firing, the starting time is the maximum of the two starting times in the bottom two cases, and therefore also equal to at least one of the two cases. This is to be expected since one could consider those events to be observable outputs of the event system, for which the same property should hold.

This property is often referred to as the *superposition property* and can be effectively used to simplify the analysis of the response of a system to a complex combination of stimuli.

Example C.10 (Superposition in the Railroad Network). The railroad network of Example C.5 can be seen as an event system if the inputs to the system are trains that are arriving from Liège. Trains departing from Maastricht need to wait for them to depart. The outputs of the system are trains arriving at the station Schiphol. The railroad network is a max-plus-linear index-invariant system since it is modelled as a dataflow graph. We can therefore apply the superposition property to study the impact of individual events in an event sequence. Assume that the top graph in Figure C.17 represents the regular schedule of the trains according to the inputs

$$l = [0, 5, 10, 15, 20, 25]$$

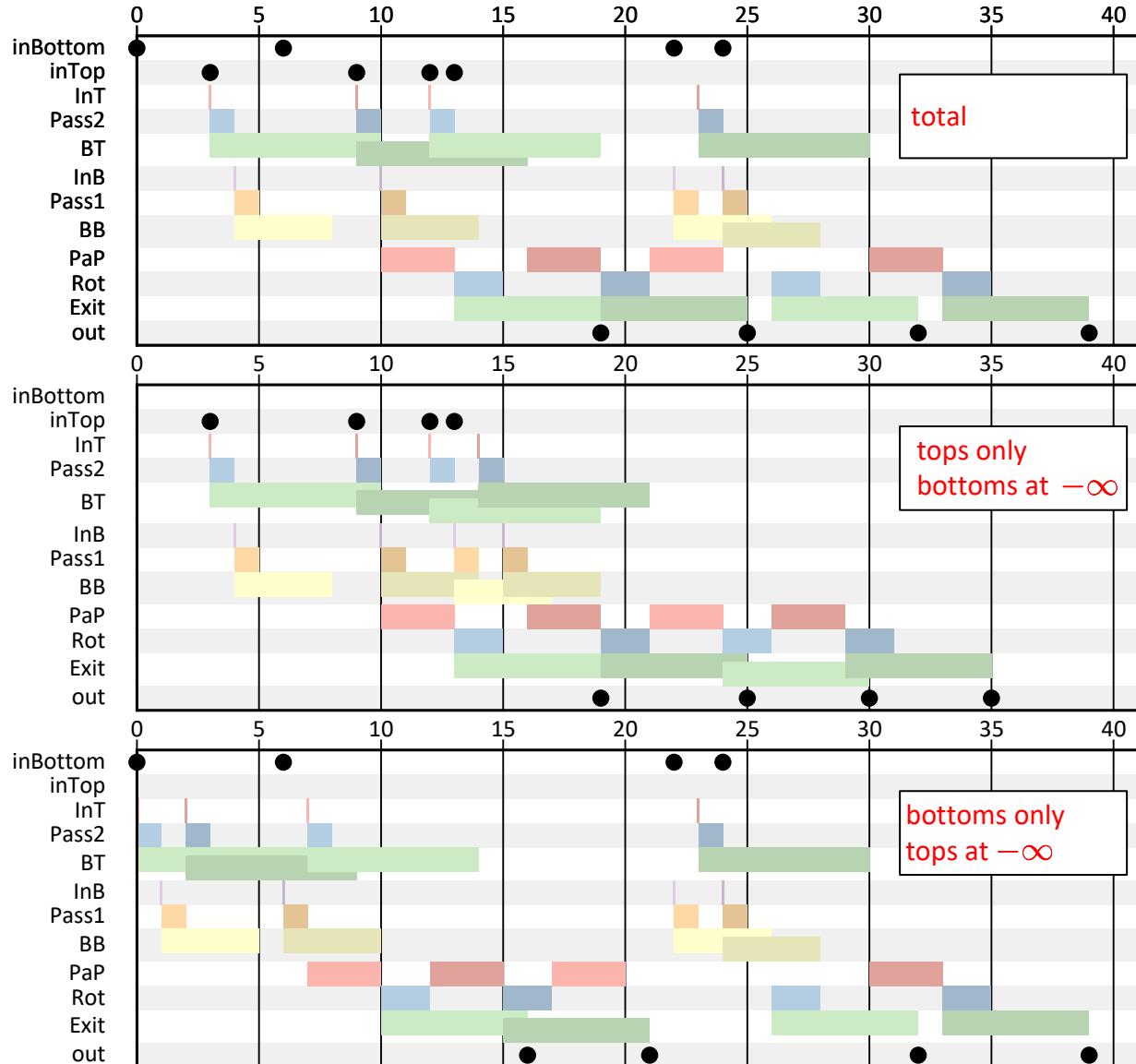


Figure C.16: Superposition in the manufacturing system

As outputs of the system, assuming self-timed execution with trains (tokens) T_1 through T_4 available at time 0, trains are arriving at Schiphol according to the sequence

$$s = [4, 8, 12, 16, 21, 26]$$

Imagine that we are informed that the second train from Liège, scheduled for 5.00hrs will instead arrive, delayed, at 7hrs. We now want to predict how many of the trains to Schiphol are impacted by this delay. We find the simplest event sequence d such that $l_d = l \oplus d$ corresponds to the delayed event sequence. The only event that has

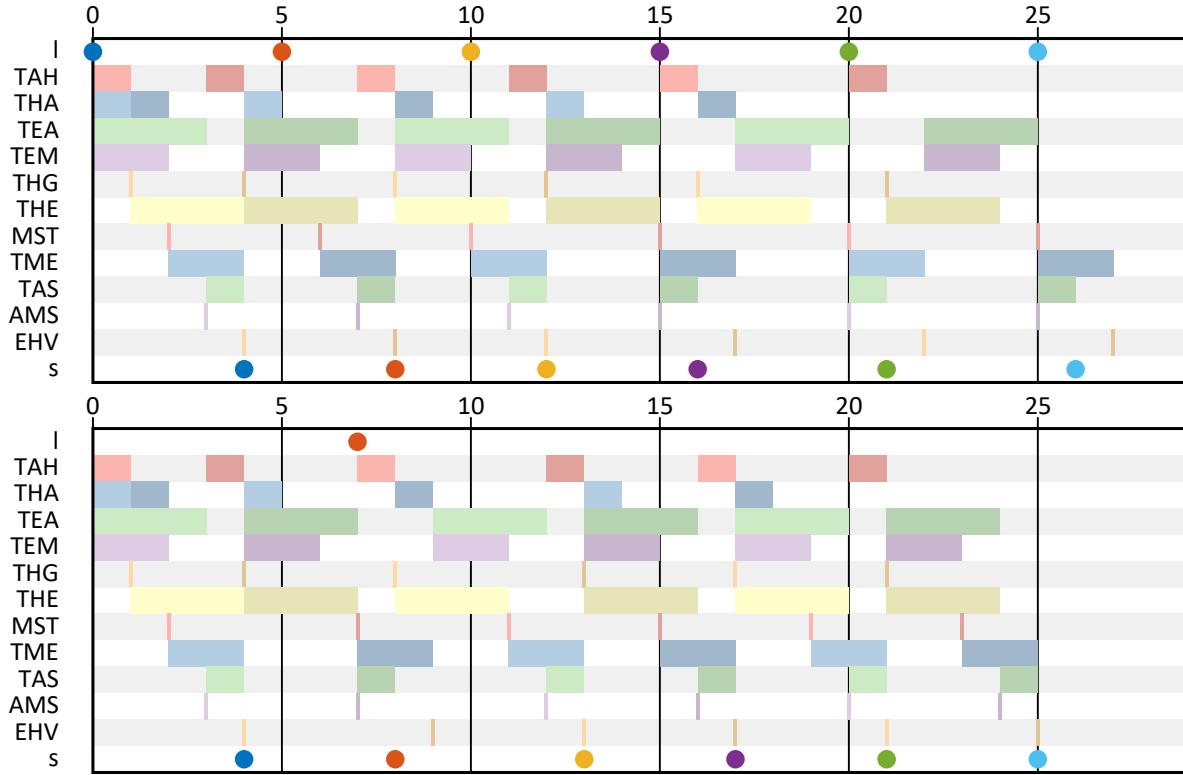


Figure C.17: Superposition of events in the railroad network

changed is $l(1)$, so we can take

$$d = [-\infty, 7, -\infty, -\infty, -\infty, -\infty]$$

and determine the response of the system to input d . This response is shown in the bottom part of Figure C.17. The real arrivals at Schiphol will be according to the maximum of both sequences l and d . We can see that the third and the fourth train at Schiphol will be delayed by one hour. The trains after that will arrive at their scheduled times.

Exercise C.11 (The manufacturing system – superposition). Consider the dataflow model of the running example of the manufacturing system of Example C.2 as a max-plus-linear event system.

1. What is the response of the system to an input sequence $[0, 2, 8, 17, 24]$ of tops?
2. What is the response of the system to an input sequence $[4, 11, 14, 20, 27]$ of bottoms?
3. Use superposition to compute the response of the system to a combined input $[0, 2, 8, 17, 24]$ of tops and $[4, 11, 14, 20, 27]$ of bottoms.

4. What is the effect on the output production times and the makespan of producing five products of the first bottom arriving 1 time unit late? What is the best possible makespan for producing five products when the first bottom arrives at time 4? How much can this makespan be improved by providing the first bottom earlier? How much can the arrival of the first bottom be further delayed without affecting the optimal makespan?

See answer

C.5 Impulse Response

Linearity of an event system is an important property. It follows directly from the linearity properties that linear event systems allow for a number of convenient analysis techniques. An example is the use of the *superposition property*, as discussed in the previous section. To further investigate this property we define an event sequence that represents an atomic, individual stimulus. This is the *impulse sequence*.

Definition C.18 (Impulse Sequence). The *impulse sequence* is the event sequence $\delta : \mathbb{N} \rightarrow \mathbb{T}$ such that $\delta(0) = 0$ and $\delta(k) = -\infty$ for $k > 0$.

The impulse sequence is different from the zero element ($-\infty$) of the \otimes operator only for the first index $k = 0$, where it is equal to the unit element (0) of the \otimes operator. Note that the impulse event sequence is a prominent example of an event sequence with decreasing time stamps: $\delta(k) < \delta(0)$ for all $k > 0$.

A linear and index-invariant system exposes *all of its behavior* by its response to the impulse sequence.

Definition C.19 (Impulse Response). The *impulse response* of an event system S is the output event sequence o , such that $\delta \xrightarrow{S} o$, where δ is the impulse sequence.

The concept of an impulse response can also be applied to systems with multiple inputs and multiple outputs. The superposition property guarantees us independence also between events on different system inputs. Hence, the impulse response of a system S on input k and output m is o_m if:

$$\underbrace{\epsilon, \dots, \epsilon}_{k-1}, \delta, \epsilon, \dots, \epsilon \xrightarrow{S} o_1, \dots, o_m, \dots, o_l$$

Example C.11 (Conveyor Belt Impulse Response). The impulse response of the conveyor belt is the output sequence that corresponds to the input impulse sequence. All objects are present as early as needed ($-\infty$), except for object 0, which arrives at

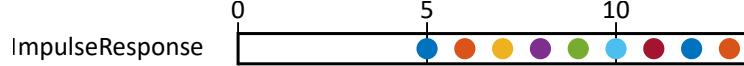


Figure C.18: Conveyor belt impulse response.

time 0. Hence, the output sequence, the *impulse response* of the belt is the sequence

$$h = \left(\frac{l}{v}, \frac{l+d}{v}, \frac{l+2d}{v}, \frac{l+3d}{v}, \dots \right).$$

The impulse response is shown in Figure C.18 for $l = 5$, $v = 1$ and $d = 1$. As all objects following object 0 are placed on the belt as soon as possible, one after the other, after each object a delay of $\frac{d}{v}$ occurs after which there is room for a new object. Now imagine a particular arrival of input objects at time stamps o_1 , o_2 and o_3 .

$$(o_1, o_2, o_3, -\infty, -\infty, \dots) = o_1 \otimes \delta \oplus o_2 \otimes \delta(k-1) \oplus o_3 \otimes \delta(k-2)$$

Linearity and index invariance tell us immediately that the corresponding output sequence is

$$a = o_1 \otimes h \oplus o_2 \otimes h(k-1) \oplus o_3 \otimes h(k-2).$$

We can write any input sequence i as a linear combination of delayed impulses as follows:

$$i(k) = \bigoplus_{0 \leq m \leq k} i(m) \otimes \delta(k-m) = \bigoplus_{0 \leq m \leq k} i(m) \otimes \delta^m(k) = \bigoplus_{m \geq 0} i(m) \otimes \delta^m(k)$$

Hence,

$$i = \bigoplus_{m \geq 0} i(m) \otimes \delta^m$$

Since $\delta \xrightarrow{S} h$, by linearity and index invariance, we know that:

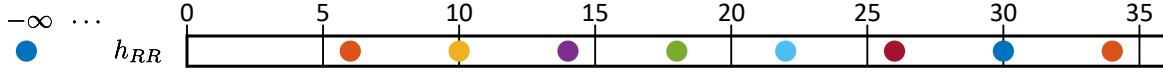
$$o = \bigoplus_{m \geq 0} i(m) \otimes h^m$$

Which means that element k of the output sequence o is equal to:

$$o(k) = \bigoplus_{0 \leq m \leq k} i(m) \otimes h(k-m) \tag{C.1}$$

The operation on two event sequences that we recognize on the right-hand side of Equation C.1 is called the *convolution* of the event sequences i and h . The result of the operation is a new event sequence. It is denoted $i \otimes h$.

Definition C.20 (Convolution). Let s and t be two event sequences. Their *convolution* is defined as

Figure C.19: Impulse response h_{RR} of the railroad network.

lution, $s \otimes t$, is the event sequence defined as follows.

$$(s \otimes t)(k) = \bigoplus_{0 \leq m \leq k} s(m) \otimes t(k-m)$$

Note that, by definition of the convolution operation, it is possible to compute a finite prefix (an initial part) of the infinite event sequence that is defined as the convolution of two infinite event sequences from finite prefixes of those event sequences. To compute a prefix of length k of the convolution, prefixes of length k of both sequences are needed.

A linear and index-invariant system is entirely defined by its impulse response. No other information about the system is required to determine its output from its input.

Theorem C.4 (Convolution). If S is a max-plus-linear and index-invariant system with impulse response h , then for any input event sequence i , $i \xrightarrow{S} i \otimes h$.

Note that event with index k in the convolution does not depend on events with indices larger than k in either of the functions from which it is computed. This means that in a linear and index-invariant system, output events do not depend on future input events.

The theorem is stated for an event system with a single input and a single output, but it can be easily generalized to multiple inputs and outputs. If S is a linear and index-invariant event system with k inputs and m outputs, with $h_{l,n}$ the impulse response from input l to output n , then

$$i_1, \dots, i_k \xrightarrow{S} o_1, \dots, o_m$$

where for $1 \leq n \leq m$,

$$o_n = \bigoplus_{l=1}^k i_l \otimes h_{l,n}$$

Example C.12 (Railroad Network, Impulse Response, Convolution). Recall that the railroad network of Examples C.5 and C.10 can be seen as a max-plus-linear index-invariant event system with as inputs the trains that are arriving from Liège and outputs the trains arriving at Schiphol if we assume that the first trains that do not depend on any input, T_1 through T_4 , depart at time $-\infty$. Its impulse response is equal to

$$h_{RR} = [-\infty, 6, 10, 14, 18, 22, \dots]$$

The impulse response is visualized in Figure C.19. The k -th element of an impulse response of a system modeled by a dataflow graph is the duration of the longest path with k tokens from the respective input to the respective output. If we look at the

dataflow graph, we see that there is no path from Liège to Schiphol through channels without tokens. Hence, the first output $h_{RR}(0) = -\infty$ does not depend on the impulse. The initial trains have departed at time $-\infty$ and the first train arrives at the output at $h_{RR}(0) = -\infty$. The second event at the output has time stamp 6. The longest (and only) path from Liège to Schiphol with one token on the path is six hours. It continues in such a way that $h_{RR}(k) = 2 + 4k$ for all $k \geq 1$, the longest path from input to output with k tokens on it.

Let's reconsider Example C.10. The scheduled arrival times of trains from Liège given in that example are $l = [0, 5, 10, 15, 20, 25]$. The scheduled arrival times $s = [4, 8, 12, 16, 21, 26]$ at Schiphol were given assuming self-timed execution and availability of the initial trains T_1 through T_4 at time 0. We can now reconstruct these arrival times analytically, using the above impulse response and, through Theorem C.4, convolution. Since the impulse response computed above assumes that initial trains (tokens) have time stamp $-\infty$, we take the effect of the availability of initial trains at time 0 into account through superposition. (Alternatively, we could have transformed the model as explained below Theorem C.3. We come back to this alternative in Exercise C.12.) Consider the dataflow graph of Figure C.7. Assume all input trains from Liège are available at time $-\infty$ and that the initial trains (tokens) are available at time 0. It is then not difficult to deduce that the arrival times $s_{st}(k)$ at Schiphol under self-timed execution are $4 + 4k$, for $k \geq 0$.

According to Theorem C.4 and using superposition, $s = l \otimes h_{RR} \oplus s_{st}$. This gives the following results for the arrival times of trains at Schiphol:

$$\begin{aligned} s(0) &= (l \otimes h_{RR})(0) \oplus s_{st}(0) \\ &= l(0) \otimes h_{RR}(0) \oplus 4 = 0 \otimes -\infty \oplus 4 = -\infty \oplus 4 = 4 \\ s(1) &= (l \otimes h_{RR})(1) \oplus s_{st}(1) \\ &= l(0) \otimes h_{RR}(1) \oplus l(1) \otimes h_{RR}(0) \oplus 8 = 6 \oplus -\infty \oplus 8 = 8 \\ s(2) &= (l \otimes h_{RR})(2) \oplus s_{st}(2) \\ &= l(0) \otimes h_{RR}(2) \oplus l(1) \otimes h_{RR}(1) \oplus l(2) \otimes h_{RR}(0) \oplus 12 \\ &= 0 \otimes 10 \oplus 5 \otimes 6 \oplus 10 \otimes -\infty \oplus 12 = 10 \oplus 11 \oplus -\infty \oplus 12 = 12 \\ s(3) &= (l \otimes h_{RR})(3) \oplus s_{st}(3) \\ &= 0 \otimes 14 \oplus 5 \otimes 10 \oplus 10 \otimes 6 \oplus 15 \otimes -\infty \oplus 16 = 14 \oplus 15 \oplus 16 \oplus -\infty \oplus 16 = 16 \\ s(4) &= (l \otimes h_{RR})(4) \oplus s_{st}(4) \\ &= 0 \otimes 18 \oplus 5 \otimes 14 \oplus 10 \otimes 10 \oplus 15 \otimes 6 \oplus 20 \otimes -\infty \oplus 20 = 21 \\ s(5) &= (l \otimes h_{RR})(5) \oplus s_{st}(5) \\ &= 0 \otimes 22 \oplus 5 \otimes 18 \oplus 10 \otimes 14 \oplus 15 \otimes 10 \oplus 20 \otimes 6 \oplus 25 \otimes -\infty \oplus 24 = 26 \end{aligned}$$

We may observe that initially the availability of the initial trains determines the arrivals at Schiphol, i.e., the maximum is found in s_{st} , whereas the arrival of the last two trains is determined by the input arrivals, the maximum is found in $l \otimes h_{RR}$.

Exercise C.12 (A wireless channel decoder – impulse responses). Reconsider the fragment of the Wireless Channel Decoder (WCD) dataflow model of Exercise

C.9. In Exercise C.10, we showed that this graph specifies a max-plus-linear index-invariant system under a simplifying assumption on the production times of tokens on the self-loop of Actor **Sh**. We can now compute the self-timed output production times analytically.

We constrain the earliest firing of Actor **Sh** in self-timed execution to time 0. Theorem C.3 states that a dataflow graph with initial tokens available at time $-\infty$ is a max-plus-linear index-invariant system. As explained with Theorem C.3 and illustrated with Figure C.15 we may consider constraints on the initial token availability in the form of an additional input. We do so in this case with an input called *sh* to actor **Sh** that prevents it from starting earlier than the self-timed execution allows. We moreover assume that the initial token on the self-loop is available at time $-\infty$. We then obtain another max-plus-linear system that is index invariant. Based on Theorem C.4, this allows us to compute the (self-timed) output production using convolutions with the impulse responses for each of the inputs. We use indices as in the text earlier to distinguish different impulse responses.

1. What is the event sequence that should be offered to the additional input *sh* to capture the constraint that the earliest firing time of the first firing of actor **Sh** is 0?
2. What are the impulse responses $h_{i,o}$, $h_{ce,o}$, and $h_{sh,o}$?
3. Provide an analytic expression for event sequence o in terms of these impulse responses.
4. Compute the first three outputs on o assuming self-timed execution and input arrivals $i(k) = ce(k) = 4k$, for $k \geq 0$.

See answer

C.6 Max-Plus Linear Algebra

C.6.1 Definitions

Equations in max-plus algebra describe relations and systems that are linear. Just like traditional linear systems, it is convenient to use *matrix* and *vector* notation to compactly represent sets of equations and transformations involving multiple variables.

Definition C.21 (Max-Plus Vector). A max-plus vector of size n is a tuple $\mathbf{v} \in \mathbb{T}^n$ of n time stamps. It is written as a column of values enclosed by square brackets, e.g.,

$$\begin{bmatrix} 0 \\ -\infty \end{bmatrix} \in \mathbb{T}^2$$

- Variables that represent max-plus vectors are written as bold, roman lower case letters, e.g., \mathbf{v} .
- $\mathbf{v}[k]$, $1 \leq k \leq m$ refers to the k -th time stamp in the vector.
- We use ϵ to denote a vector containing only the value $-\infty$ and $\mathbf{0}$ to denote a vector containing only the value 0. In both cases the size of the vector should be clear from the context.
- We use \mathbf{i}_k to denote a vector \mathbf{v} such that $\mathbf{v}[k] = 0$ and $\mathbf{v}[m] = -\infty$ for $m \neq k$.

If \mathbf{v} and \mathbf{w} are two vectors of the same size, we can compute their *maximum* by applying the additive operator \oplus to corresponding elements, i.e., $(\mathbf{v} \oplus \mathbf{w})[k] = \mathbf{v}[k] \oplus \mathbf{w}[k]$, the result is a vector, $\mathbf{v} \oplus \mathbf{w}$, also of the same size. Recall that, according to the \oplus operator we take the element-wise maximum of the corresponding vector elements.

A vector \mathbf{v} can also be *delayed* by a scalar constant $c \in \mathbb{T}$. The result is a new vector, $c \otimes \mathbf{v}$, which is such that $(c \otimes \mathbf{v})[k] = c \otimes (\mathbf{v}[k])$. That means, we get a new vector of the same length where the value c has been added to each of the elements of the original vector \mathbf{v} . If one vector is a delayed version of the other, the vectors are called *dependent*. Otherwise they are *independent*.

There is a strong analogy between classical linear algebra and max-plus linear algebra. We can also define an inner product between vectors \mathbf{v} and \mathbf{w} of the same size n . $(\mathbf{v}, \mathbf{w}) = \bigoplus_{1 \leq k \leq n} \mathbf{v}[k] \otimes \mathbf{w}[k]$. It is easy to see that $(\mathbf{v}, \mathbf{w}) = (\mathbf{w}, \mathbf{v})$. It is often written as $\mathbf{v}^T \mathbf{w}$ because of the similarity with matrix multiplication.

The *maximum* and *delay* on max-plus vectors \mathbf{v} and \mathbf{w} are defined as follows:

$$(\mathbf{v} \oplus \mathbf{w})[k] = \mathbf{v}[k] \oplus \mathbf{w}[k]$$

$$(c \otimes \mathbf{v})[k] = c \otimes (\mathbf{v}[k])$$

Vectors \mathbf{v} and \mathbf{w} are called *dependent* if there exists $c \in \mathbb{T}$ such that $c \otimes \mathbf{v} = \mathbf{w}$ or $\mathbf{v} = c \otimes \mathbf{w}$.

The *inner product* (\mathbf{v}, \mathbf{w}) between vectors \mathbf{v} and \mathbf{w} is

$$(\mathbf{v}, \mathbf{w}) = \bigoplus_{1 \leq k \leq n} \mathbf{v}[k] \otimes \mathbf{w}[k].$$

Definition C.22 (Max-Plus Matrix). A max-plus matrix of size n by m is a table $\mathbf{A} \in \mathbb{T}^{n \times m}$ of n rows and m columns containing time stamps. It is written as a table of values enclosed by square brackets, e.g.,

$$\begin{bmatrix} 0 & -\infty \\ -\infty & 0 \end{bmatrix} \in \mathbb{T}^{2 \times 2}$$

- Variables that represent max-plus matrices are written as bold, roman upper

case letters, e.g., \mathbf{A} .

- $\mathbf{A}[k, m]$ refers to the time stamp in the matrix at row k and column m .
- If $\mathbf{A} \in \mathbb{T}^{n \times m}$ is a max-plus matrix, we use \mathbf{A}^T to refer to the matrix in $\mathbb{T}^{m \times n}$ such that $\mathbf{A}^T[k, l] = \mathbf{A}[l, k]$. \mathbf{A}^T is called the *transposed* of \mathbf{A} .

Definition C.23 (Max-Plus Matrix-Vector Multiplication). A max-plus matrix $\mathbf{A} \in \mathbb{T}^{m \times n}$ can be multiplied with a vector $\mathbf{v} \in \mathbb{T}^n$. The result is a vector $\mathbf{w} \in \mathbb{T}^m$ such that

$$\mathbf{w}[k] = \bigoplus_{1 \leq i \leq n} \mathbf{A}[k, i] \otimes \mathbf{v}[i]$$

It is written as $\mathbf{w} = \mathbf{Av}$.

Definition C.24 (Max-Plus Matrix-Matrix Multiplication). A max-plus matrix $\mathbf{A} \in \mathbb{T}^{m \times n}$ can be multiplied with a matrix $\mathbf{B} \in \mathbb{T}^{n \times k}$. The result is a matrix $\mathbf{C} \in \mathbb{T}^{m \times k}$ such that

$$\mathbf{C}[i, j] = \bigoplus_{1 \leq l \leq n} \mathbf{A}[i, l] \otimes \mathbf{B}[l, j]$$

It is written as $\mathbf{C} = \mathbf{AB}$.

A vector norm that is convenient in max-plus linear algebra is the *max norm*, a measure that assigns to a vector the maximum of the elements in the vector.

Definition C.25 (Max-Plus Vector Norm). The (max) norm of a max-plus vector \mathbf{v} of size n is

$$|\mathbf{v}| = \bigoplus_{1 \leq k \leq n} \mathbf{v}[k]$$

A vector \mathbf{v} is called *normal* if $|\mathbf{v}| = 0$.

Definition C.26 (Max-Plus Vector Normalization). If \mathbf{v} is a max-plus vector with a norm $|\mathbf{v}| \neq -\infty$, then the *normalized vector* of \mathbf{v} is the vector $(-|\mathbf{v}|) \otimes \mathbf{v}$. The norm of a normalized vector is 0, i.e., it is normal.

Note that a vector \mathbf{v} with norm $-\infty$ cannot be normalized, as there exists no constant $c \in \mathbb{T}$ such that $|c \otimes \mathbf{v}| = 0$. A normalized vector has no positive entries. All of its entries are 0, negative numbers, or $-\infty$.

Matrix-vector and matrix-matrix multiplication in max-plus algebra are linear. For all $c, d \in \mathbb{T}$, vectors \mathbf{x}, \mathbf{y} and matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, the following equations hold (provided the matrices and vectors are of appropriate sizes for the operations performed)

$$\begin{aligned} \mathbf{A}(c \otimes \mathbf{x} \oplus d \otimes \mathbf{y}) &= c \otimes (\mathbf{Ax}) \oplus d \otimes (\mathbf{Ay}) \\ \mathbf{A}(c \otimes \mathbf{B} \oplus d \otimes \mathbf{C}) &= c \otimes (\mathbf{AB}) \oplus d \otimes (\mathbf{AC}) \end{aligned}$$

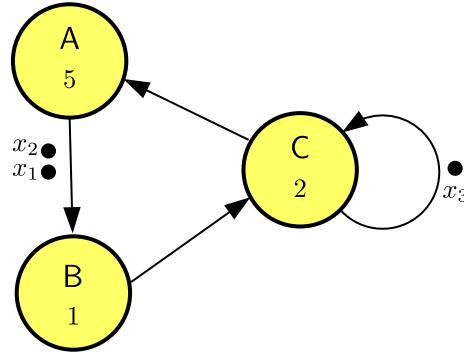


Figure C.20: Example of a timed dataflow model.



Figure C.21: Sequence of token time stamps.

C.6.2 Modelling Dataflow Graphs with Max-Plus Matrices

Dataflow graphs are max-plus-linear systems and their behavior can be expressed in max-plus algebra. In particular, the time stamps of events, such as input arrivals, token productions, actor firings and output productions can be related to each other with equations in max-plus algebra. Such sets of equations are conveniently captured using matrices and vectors. We illustrate this with the following example.

Example C.13 (Matrix Equation of a Dataflow Graph). Consider the dataflow graph in Figure C.20. It has three initial tokens. We assume the tokens have time stamps indicating their first availability times. These time stamps are indicated in Figure C.20 as x_1 , x_2 and x_3 . The tokens time stamped x_1 and x_2 are on the same channel, such that the token with time stamp x_1 is the first to be consumed by actor B. We know that after one firing of each of the actors the configuration of tokens on the channels will return to the same state, but with new time stamps. We can derive equations for those new time stamps under self-timed execution as follows. Actor B will fire at time x_1 . This firing completes at time $x_1 + 1 = 1 \otimes x_1$. The firing of actor C needs the token produced by B as well as the token on its self-edge with time stamp x_3 . Therefore, C starts its firing at $\max(1 \otimes x_1, x_3) = 1 \otimes x_1 \oplus x_3$ and completes the firing 2 time units later, at $3 \otimes x_1 \oplus 2 \otimes x_3$. At this time, the new token on the self-edge of C is produced, so it gets the time stamp $x'_3 = 3 \otimes x_1 \oplus 2 \otimes x_3$. Finally, actor A starts its firing at the same time that C finishes and A itself finishes 5 time units later at $8 \otimes x_1 \oplus 7 \otimes x_3$. Therefore, we have $x'_2 = 8 \otimes x_1 \oplus 7 \otimes x_3$. The token with time stamp x_2 was not consumed in the process, but it has moved up to the position of the first token to be consumed next by B; therefore, we have $x'_1 = x_2 = 0 \otimes x_2$. Summarizing,

we have the following equations.

$$\begin{aligned}x'_1 &= 0 \otimes x_2 \\x'_2 &= 8 \otimes x_1 & 7 \otimes x_3 \\x'_3 &= 3 \otimes x_1 & 2 \otimes x_3\end{aligned}$$

We have formatted the equations above in such a way that it is easy to see that we can collect these equations in the form of a matrix-vector equation:

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} -\infty & 0 & -\infty \\ 8 & -\infty & 7 \\ 3 & -\infty & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Note how in case there is no dependency from an old time stamp to a new time stamp, we fill the matrix with $-\infty$ at the corresponding location. Considering the fact that the dataflow graph actors keep firing over and over again, we can associate with the sequences of time stamps they produce the event sequences $x_1(k)$, $x_2(k)$ and $x_3(k)$. The equation can then recursively be used to compute the sequence of all tokens as follows.

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \\ x_3(k+1) \end{bmatrix} = \begin{bmatrix} -\infty & 0 & -\infty \\ 8 & -\infty & 7 \\ 3 & -\infty & 2 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \\ x_3(k) \end{bmatrix}$$

In short, with $\mathbf{x}(k) = \begin{bmatrix} x_1(k) \\ x_2(k) \\ x_3(k) \end{bmatrix}$ and $\mathbf{G} = \begin{bmatrix} -\infty & 0 & -\infty \\ 8 & -\infty & 7 \\ 3 & -\infty & 2 \end{bmatrix}$, we have

$$\mathbf{x}(k+1) = \mathbf{G}\mathbf{x}(k)$$

The resulting sequences of time stamps are shown in Figure C.21.

We can use the result, for example, to answer questions about makespan, such as what is the makespan of the self-timed execution of 6 iterations of the dataflow graph? In this example, we know that the last firing completes with the production of a token, so we can express the makespan as

$$M = |\mathbf{G}^6 \mathbf{0}| = 26$$

The result can be verified on Figure C.21 as the norm of the last (red) vector.

Exercise C.13 (A producer-consumer pipeline – max-plus matrix equation).

Reconsider the producer-consumer pipeline of Exercise C.2 with function f_2 for the filter actor F .

1. Provide a max-plus-matrix equation for this dataflow graph.
2. Compute the makespan of one, two, and three iterations of the graph, assuming self-timed execution with all initial tokens available at time 0.

See answer

C.7 Converting Dataflow Graphs to Max-Plus Matrices

In Example C.13 and Exercise C.13, we have derived a matrix equation that captures the relation between subsequent vectors of time stamps of the initial tokens of the graph in subsequent iterations of the dataflow graph. In both cases, all the relevant events have time stamps that can be expressed as max-plus-linear combinations of the time stamps of the initial tokens in the dataflow graph. This is in fact true in general and we exploit this property to develop a general method to compute the max-plus matrix for a deadlock-free single-rate or consistent multi-rate dataflow graph.

Definition C.27 (Symbolic Time Stamp). Let x_1, x_2, \dots, x_n be a finite set of *variables* that represent time stamps. An expression of the form

$$a_1 \otimes x_1 \oplus a_2 \otimes x_2 \oplus \dots \oplus a_n \otimes x_n$$

that represents a max-plus-linear combination of the given variables for *constants* $a_1, a_2, \dots, a_n \in \mathbb{T}$ is called a *symbolic time stamp*.

A symbolic time stamp is referred to as *symbolic*, because it represents a time stamp, but not as a value, but as an expression in terms of variables. Note that any symbolic time stamp that depends on only a strict subset of the variables can still be expressed as a symbolic time stamp on the full set of variables using $-\infty$ as the corresponding coefficient for any of the unused variables. For example, $2 \otimes x_2 = -\infty \otimes x_1 \oplus 2 \otimes x_2$.

We are often working with a number of symbolic time stamps referring to the same, fixed, set of variables, in which case only the coefficients vary. The coefficients can be conveniently collected in a vector.

Definition C.28 (Symbolic Time-Stamp Vector). A *symbolic time-stamp vector* for a set x_1, x_2, \dots, x_n of variables is a vector $\mathbf{t} \in \mathbb{T}^n$. It *represents* the symbolic time stamp

$$\mathbf{t}[1] \otimes x_1 \oplus \dots \oplus \mathbf{t}[n] \otimes x_n$$

Note that the symbolic time stamp represented by the symbolic time-stamp vector \mathbf{t} can be expressed using the vector inner product as $(\mathbf{t}, [x_1 \dots x_n]^T)$.

Various operations on symbolic time stamps can be performed directly on the symbolic time-stamp vectors that represent them. The results we need are presented below.

Let \mathbf{t}_1 be a symbolic time-stamp vector for the symbolic time stamp e_1 on the time stamp variables x_1, \dots, x_n , \mathbf{t}_2 be a symbolic time-stamp vector for the symbolic time stamp e_2 and $c \in \mathbb{T}$. Then the following results hold.

- $\mathbf{t}_1 \oplus \mathbf{t}_2$ is the symbolic time-stamp vector for the symbolic time stamp $e_1 \oplus e_2$
- $c \otimes \mathbf{t}_1$ is the symbolic time-stamp vector for the symbolic time stamp $c \otimes e_1$
- $\mathbf{i}_1 = [0 \ -\infty \ -\infty \ \dots]^T$ is the symbolic time-stamp vector for the symbolic time stamp x_1
- $\mathbf{i}_2 = [-\infty \ 0 \ -\infty \ -\infty \ \dots]^T$ is the symbolic time-stamp vector for the symbolic time stamp x_2
- \dots
- $\mathbf{i}_n = [-\infty \ \dots \ -\infty \ 0]^T$ is the symbolic time-stamp vector for the symbolic time stamp x_n

We can now define a general method to compute for a single-rate dataflow graph a max-plus matrix that expresses the relation between the time stamps of the tokens in the graph before and after the execution of an iteration of the graph (one firing of each of the actors). It is a generalization of the method we used in Example C.13 and Exercise C.13.

The max-plus matrix for a single-rate dataflow graph *without inputs* can be constructed with the following procedure. Note that outputs do not influence the behavior of the graph and do not need to be considered. We will see in Section C.10.2 how inputs and outputs can be included.

Algorithm C.1 (Symbolic Simulation). Perform the following steps.

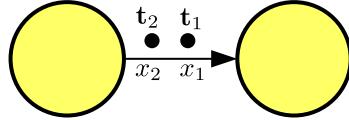
- Introduce a time-stamp variable x_k , $1 \leq k \leq n$ for each of the n initial tokens in the dataflow graph.
- Assign a symbolic time stamp $e_k = x_k$ to each of the initial tokens, represented by a symbolic time stamp vector \mathbf{i}_k .
- While there is an actor that is enabled and has not fired do the following.
 - Select an arbitrary actor a that is enabled and has not fired yet in the symbolic simulation.
 - Determine the symbolic time-stamp vector for the tokens that a produces after consuming tokens with symbolic time-stamp vectors $\mathbf{t}_1, \dots, \mathbf{t}_m$ as follows.

$$\mathbf{t}_p = e(a) \otimes \bigoplus_{1 \leq i \leq m} \mathbf{t}_i$$

- Create the matrix \mathbf{G} for the dataflow graph by collecting the symbolic time-stamp vectors for all the tokens, $\mathbf{t}_1, \dots, \mathbf{t}_n$, in the graph (they have returned to their original positions) in the order in which the variables x_1 to x_n were allocated and set row number k of matrix \mathbf{G} equal to \mathbf{t}_k^T .

Note that during the procedure, it is always possible to find an actor that is enabled and has not fired until all actors have fired if and only if the graph is deadlock free.

It may happen that some initial tokens in the graph have not been produced or consumed during the symbolic simulation. In particular, this happens precisely on the channels with two or more initial tokens from which one will be consumed and onto which one new token will be produced, like on the example channel below (and in Example C.13). In this case, x_1 is consumed during the simulation, but x_2 is not. A new token is also produced onto the channel during the simulation. The token x_2 from the initial state has now become x_1 . The newly produced token becomes the new x_2 . Therefore, the symbolic time stamp \mathbf{t}_1 for x_1 at the end of the simulation is equal to the initial symbolic time stamp \mathbf{t}_2 of x_2 at the start of the simulation, i.e., \mathbf{i}_2 .



During symbolic simulation, we can record the symbolic time stamps and symbolic time-stamp vectors for any of the events that occur during the self-timed execution of the dataflow graph, not only for the produced tokens. In particular, we can record the start times and completion times of actor firings. The actor firing that consumes the tokens with symbolic time-stamp vectors $\mathbf{t}_1, \dots, \mathbf{t}_m$ starts its firing at the time represented by:

$$\mathbf{t}_a^{start} = \bigoplus_{1 \leq i \leq m} \mathbf{t}_i$$

The symbolic time-stamp vector of the completion of an actor firing is:

$$\mathbf{t}_a^{completion} = e(a) \otimes \mathbf{t}_a^{start}$$

During the symbolic-simulation procedure we can therefore also produce the matrices \mathbf{H}^s and \mathbf{H}^c , where the rows of \mathbf{H}^s are the symbolic time-stamp vectors of all actor-firing start times and similarly the rows of \mathbf{H}^c are the symbolic time-stamp vectors of all actor-firing completion times.

If we know the concrete time stamps of the tokens in the graph in the form of the vector \mathbf{x} , we can directly compute a vector \mathbf{s} with all the concrete actor-firing start times and a vector \mathbf{c} with all the concrete actor-firing completion times in an iteration as:

$$\mathbf{s} = \mathbf{H}^s \mathbf{x}, \quad \mathbf{c} = \mathbf{H}^c \mathbf{x}$$

And as we use the max-plus matrix \mathbf{G} to compute the sequence $\mathbf{x}(k)$ of time stamp-vectors using the equation $\mathbf{x}(k+1) = \mathbf{G}\mathbf{x}(k)$, we can compute all actor-firing start

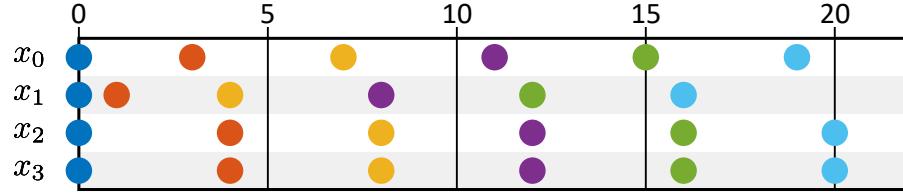


Figure C.22: Sequence of time-stamp vector of the self-timed execution of the railroad network

and completion times as:

$$\mathbf{s}(k) = \mathbf{H}^s \mathbf{x}(k), \quad \mathbf{c}(k) = \mathbf{H}^c \mathbf{x}(k)$$

We can use this, for example, to generate a Gantt-chart visualization [1], as it is done in the CMWB.

Example C.14 (Railroad Network, Symbolic Simulation). In the railroad-network dataflow graph of Figure C.7, we have four initial tokens T_1, \dots, T_4 . We label these with the symbolic time-stamp vectors $\mathbf{i}_1, \dots, \mathbf{i}_4$.

For this example, we ignore the input and output of the network. During the symbolic simulation, we can execute the actors in the order given in the table below. We find the corresponding symbolic time-stamp vectors for the actor firing starts and completions.

| actor | symbolic start | symbolic completion |
|-------|---------------------------------|---------------------------------|
| TEM | $[-\infty -\infty -\infty 0]^T$ | $[-\infty -\infty -\infty 2]^T$ |
| MST | $[-\infty -\infty -\infty 2]^T$ | $[-\infty -\infty -\infty 2]^T$ |
| TME | $[-\infty -\infty -\infty 2]^T$ | $[-\infty -\infty -\infty 4]^T$ |
| THA | $[-\infty 0 -\infty -\infty]^T$ | $[-\infty 1 -\infty -\infty]^T$ |
| TAH | $[0 -\infty -\infty -\infty]^T$ | $[1 -\infty -\infty -\infty]^T$ |
| THG | $[1 -\infty -\infty -\infty]^T$ | $[1 -\infty -\infty -\infty]^T$ |
| THE | $[1 -\infty -\infty -\infty]^T$ | $[4 -\infty -\infty -\infty]^T$ |
| EHV | $[4 -\infty -\infty 4]^T$ | $[4 -\infty -\infty 4]^T$ |
| TEA | $[-\infty -\infty 0 -\infty]^T$ | $[-\infty -\infty 3 -\infty]^T$ |
| AMS | $[-\infty 1 3 -\infty]^T$ | $[-\infty 1 3 -\infty]^T$ |
| TAS | $[-\infty 1 3 -\infty]^T$ | $[-\infty 2 4 -\infty]^T$ |

Token T_1 is produced by actor AMS, T_2 by THG, T_3 and T_4 by EHV. Collecting the symbolic time-stamp vectors for the new tokens (corresponding to the symbolic completion times of the corresponding actors) as the rows of the matrix gives the following result.

$$\mathbf{T} = \begin{bmatrix} -\infty & 1 & 3 & -\infty \\ 1 & -\infty & -\infty & -\infty \\ 4 & -\infty & -\infty & 4 \\ 4 & -\infty & -\infty & 4 \end{bmatrix}$$

We can use the matrix \mathbf{T} to compute the sequence $\mathbf{x}(k)$ of concrete time-stamp vectors, of the tokens in the graph, assuming that $\mathbf{x}(0) = \mathbf{0}$, i.e., that the initial trains all depart at time 0. We obtain the following result.

$$\mathbf{x}(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{x}(1) = \begin{bmatrix} 3 \\ 1 \\ 4 \\ 4 \end{bmatrix}, \quad \mathbf{x}(2) = \begin{bmatrix} 7 \\ 4 \\ 8 \\ 8 \end{bmatrix}, \quad \mathbf{x}(3) = \begin{bmatrix} 11 \\ 8 \\ 12 \\ 12 \end{bmatrix}, \quad \mathbf{x}(4) = \begin{bmatrix} 15 \\ 12 \\ 16 \\ 16 \end{bmatrix}$$

This states, for example, that the fifth train to leave from Eindhoven to Amsterdam leaves at time 16hrs.

Figure C.22 shows the sequence of time-stamp vectors obtained from the matrix \mathbf{T} and the initial vector $\mathbf{x}(0) = \mathbf{0}$.

Exercise C.14 (A producer-consumer pipeline – symbolic simulation). Re-consider the producer-consumer pipeline of Exercise C.13, with function f_2 for the filter actor F. Compute the max-plus matrix for the dataflow graph through symbolic simulation.

See answer

Exercise C.15 (An image-based control system – symbolic simulation). Consider the IBC pipeline of Exercise C.3.

1. Compute the max-plus matrix for the IBC pipeline through symbolic simulation.
2. Assume self-timed execution with initial time stamps available at time 0. What is the makespan of three actuations? What is the start time of the fifth sample period?

See answer

C.8 Monotonicity

The models that we have made and the examples that we have shown all have fixed, constant durations associated to the executed actions or to the time constraints. In reality, not all systems have this property. It often happens that there is variation in such durations. Explicitly incorporating such variation in our models may complicate modelling and/or analysis. Very often we take an alternative approach, where we assume in our models a constant *worst-case* duration that abstracts such variations. Max-plus-linear models have an important property that supports such a worst-case modelling approach, which is called *monotonicity*.

In terms of the elementary operators, this can be formulated as follows.

Monotonicity For all $x, y, z \in \mathbb{T}$:

- if $x \leq y$ then $x \oplus z \leq y \oplus z$ and $z \oplus x \leq z \oplus y$
- if $x \leq y$ then $x \otimes z \leq y \otimes z$ and $z \otimes x \leq z \otimes y$

Intuitively, for example, whenever a dataflow actor waits for the arrival of two input tokens and one of them is delayed due to our worst-case abstraction, then also the start of the actor firing (captured through a \oplus operation) is a worst-case abstraction of the actual start of the actor. Similarly, monotonicity of the \otimes operator tells us that if the firing duration of an actor firing is abstracted to a worst case, the actor completion time computed with max-plus algebra results in a valid worst-case abstraction.

Because both operators are monotone, the monotonicity result extends straightforwardly to all operations on vectors and matrices. We define less-than-or-equal for vectors and matrices by element-wise comparison.

If \mathbf{v} and \mathbf{w} are vectors of the same size n then

$$\mathbf{v} \leq \mathbf{w} \Leftrightarrow \forall_{1 \leq k \leq n} \mathbf{v}[k] \leq \mathbf{w}[k]$$

Analogously, we define the relation for matrices of the same size m by n :

$$\mathbf{A} \leq \mathbf{B} \Leftrightarrow \forall_{1 \leq k \leq m, 1 \leq l \leq n} \mathbf{A}[k, l] \leq \mathbf{B}[k, l]$$

If $c \leq c'$, $\mathbf{x} \leq \mathbf{x}'$, $\mathbf{y} \leq \mathbf{y}'$, $\mathbf{A} \leq \mathbf{A}'$, $\mathbf{B} \leq \mathbf{B}'$ we have all of the following results.

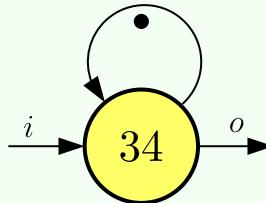
$$\begin{aligned} c \otimes \mathbf{x} &\leq c' \otimes \mathbf{x}' \\ \mathbf{x} \oplus \mathbf{y} &\leq \mathbf{x}' \oplus \mathbf{y}' \\ \mathbf{Ax} &\leq \mathbf{A}'\mathbf{x}' \\ \mathbf{AB} &\leq \mathbf{A}'\mathbf{B}' \\ |\mathbf{x}| &\leq |\mathbf{x}'| \\ (\mathbf{x}, \mathbf{y}) &\leq (\mathbf{x}', \mathbf{y}') \end{aligned}$$

We will see later how we can use max-plus algebra to compute performance metrics such as throughput or latency. The monotonicity property can be used to prove that such metrics computed on worst-case abstractions of a system results in conservative estimates of the actual system metrics.

Example C.15 (Variable Response Times). Assume that we want to model tasks executing on a processor that employs round-robin arbitration between three applications. The arbitration works as follows. It offers service of the processor to the three applications in a cyclic fashion. When the application has workload to perform, it can use the processor until all its work is done, or until it has used the processor for the maximum period P , whichever comes first. After that, or immediately, in case the application did not have any work to perform, the arbitration turns to the following application in the cycle and offers service to that application. This process continues. When none of the applications has any workload, the processor idles until one of the applications has some workload again.

From the perspective of one of the applications, the system is not deterministic. The behavior may vary with the workload of other applications. Assume the period of the round-robin arbitration is $P = 4$ and the application needs to execute tasks which need the processor for 10 time units. Then, in the best case, it takes exactly 10 time units to complete if the other applications do not require the processor service. In the worst case, it may take 34 time units. The processor may be allocated to both other applications first for two periods (8 time units) Then the 10 time units are interrupted twice, after 4 and again after 8 time units with two more interruptions of two periods each. Any duration in between 10 and 34 is also possible.

This means we could model the task execution with the following dataflow model.



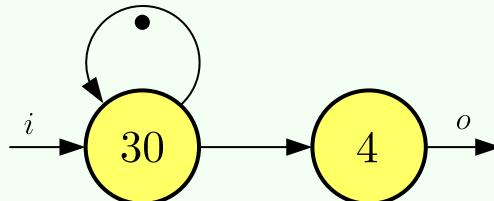
Note that the self-edge prevents multiple instances of the task from executing simultaneously, which is not possible as they would need the same processor. Hence, a deterministic worst-case model exists of the processor which is itself not deterministic. Monotonicity of the dataflow model ensures that any performance metrics we derive using the dataflow model are conservative with respect to the actual performance with the real round-robin arbitration.

The simple model we derived above is, however, more pessimistic than necessary. Imagine for example that two task instances arrive together. They need a total of 20 time units of the processor. With similar reasoning as applied above, we can conclude that this requires in the worst case, 60 time units, but the dataflow model above predicts 68 time units.

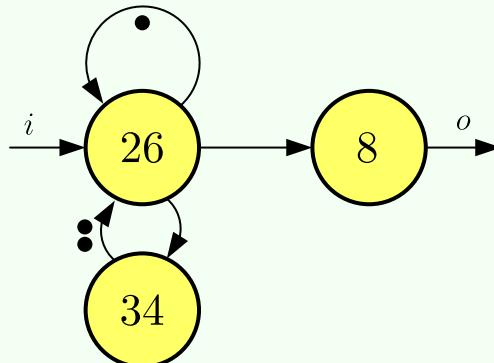
The reason is that the worst-case conditions that determine the response time of 34 for a single task cannot occur also for the second task if the processing of the second

task immediately follows the processing of the first.

We often use the following type of dataflow model instead, which is more accurate. It is called a *latency-rate* dataflow model since it has two actors, the first models the average processing rate, which is $1/30$ since the task is 10 time units and we get on average in the worst case a third of the processor. The second actor, notably without a self-edge, then accounts for the worst-case latency. For the example, it requires a delay of 4 for the latency actor to be conservative, for instance, to account for the response time of 34 for a single task.



Note, however, that this model is still pessimistic. It has, in contrast to the first model, the correct worst-case throughput, but it predicts a response time for two task executions of 64, while we have seen that it is 60. An exact worst-case model exists in this particular case, shown below, but it need not exist in general. Close approximations can be very large, so they are not commonly applied.



More details about accurate response-time modelling, especially for TDM (Time-Division Multiplex) arbiters, can be found in [17]. The key message of this example is that worst-case max-plus-linear models, because of monotonicity, can be used to reason about the performance metrics of a system under worst-case timing assumptions. These metrics then provide bounds on the actual metrics.

In the same spirit, of earlier-time-stamps-are-better, we define the following relation on event sequences that we will use in Section C.12.

Definition C.29 (Ordering Event Sequences). Let s_1 and s_2 be event sequences. $s_1 \leq s_2$ if and only if for all $k \in \mathbb{N}$ such that $s_2(k)$ is defined, $s_1(k)$ is also defined and $s_1(k) \leq s_2(k)$

Observe that in the definition of $s_1 \leq s_2$ we allow the event sequence s_1 to be longer (have more events) than the event sequence s_2 . This aligns well with our notion of worst-case modeling: a system that can produce more output events is better than a system that

does not produce those output events at all. Hence, $\epsilon \leq s$ for any event sequence s .

Convolution is monotone. If s, s', t and t' are event sequences such that $s \leq s'$ and $t \leq t'$, then

$$s \otimes t \leq s' \otimes t'$$

Therefore, max-plus-linear and index-invariant event systems are monotone. If $s \xrightarrow{S} o$ and $s' \xrightarrow{S'} o'$, then $o \leq o'$. And if system S has impulse response h and system S' has impulse response h' such that $h \leq h'$, $s \xrightarrow{S} o$ and $s \xrightarrow{S'} o'$, then $o \leq o'$.

Equations $\mathbf{Ax} = \mathbf{b}$ in max-plus algebra often do not have exact solutions. Instead, one can ask the question, what is the largest vector \mathbf{x} such that $\mathbf{Ax} \leq \mathbf{b}$ for a given matrix \mathbf{A} and vector \mathbf{b} .

For example, consider the following equation.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 7 \\ 5 \end{bmatrix}$$

We can derive the largest possible values for which the equation holds. If we write out the equation per row we obtain the following two equations.

$$\begin{aligned} 1 \otimes x_1 \oplus 2 \otimes x_2 &\leq 7 \\ 3 \otimes x_1 \oplus 4 \otimes x_2 &\leq 5 \end{aligned}$$

If we consider that $\max(a, b) \leq c$ is equivalent to $a \leq c \wedge b \leq c$, we obtain four conditions.

$$1 \otimes x_1 \leq 7, \quad 2 \otimes x_2 \leq 7$$

$$3 \otimes x_1 \leq 5, \quad 4 \otimes x_2 \leq 5$$

This means for x_1 that

$$x_1 \leq 7 - 1 \quad \text{and} \quad x_1 \leq 5 - 3$$

and for x_2 that

$$x_2 \leq 7 - 2 \quad \text{and} \quad x_2 \leq 5 - 4$$

Then the largest values for x_1 and x_2 that satisfy the constraint can easily be found to be.

$$x_1 = 2 \quad \text{and} \quad x_2 = 1$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \end{bmatrix} \leq \begin{bmatrix} 7 \\ 5 \end{bmatrix}$$

We can now generalize the method to the general case of an equation $\mathbf{Ax} \leq \mathbf{b}$. The row equation for row k is

$$\bigoplus_m \mathbf{A}[k, m] \otimes \mathbf{x}[m] \leq \mathbf{b}[k]$$

Thus we have for all k, m :

$$\mathbf{A}[k, m] \otimes \mathbf{x}[m] \leq \mathbf{b}[k]$$

Note the the equation is trivially satisfied if $\mathbf{A}[k, m] = -\infty$. Therefore, we get the equations for all k, m such that $\mathbf{A}[k, m] \neq -\infty$:

$$\mathbf{x}[m] \leq \mathbf{b}[k] - \mathbf{A}[k, m]$$

Note that in case $\mathbf{A}[k, m] = -\infty$ for all m , i.e., an entire column of \mathbf{A} is equal to $-\infty$, then $\mathbf{x}[k]$ can be arbitrarily large and such a specific largest vector \mathbf{x} does not exist. In case all columns of \mathbf{A} contain at least one value different from $-\infty$, we find the elements of the largest vector with the following equation.

$$\mathbf{x}[m] = \min_{k \text{ s.t. } \mathbf{A}[k, m] \neq -\infty} \mathbf{b}[k] - \mathbf{A}[k, m]$$

Exercise C.16 (A producer-consumer pipeline – worst-case abstractions).

Reconsider the producer-consumer pipeline of Exercise C.2, with function f_2 for the filter actor F that takes 3 time units to process once on a given processor. Assume Actor F is sharing its processor with another application. Further, assume that the processor uses TDM scheduling with a period of 4 time units, in which each application gets two of the four time slots. It is unknown which time slots are given to each of the applications, and the time slots are dedicated to the application even if it does not have any workload).

1. Provide a three-actor conservative dataflow model of the producer-consumer pipeline following the structure of the model given in Exercise C.2.
2. Adapt the model of the first item by including a conservative latency-rate abstraction for the processing of f_2 in the model.
3. What are the first three completion times of Actor C under the true worst-case conditions for F ? What are the first three completion times of C predicted by the models of the previous two items? Assume in all cases that initial tokens are available at time 0.

See answer

C.9 The Eigenvalue Equation

In traditional linear algebra, an important equation to study is the so-called *eigenvalue equation*. It is also a very important equation for max-plus-linear systems. The equation is as follows.

$$\mathbf{Ax} = \lambda \otimes \mathbf{x}, |\mathbf{x}| \neq -\infty \tag{C.2}$$

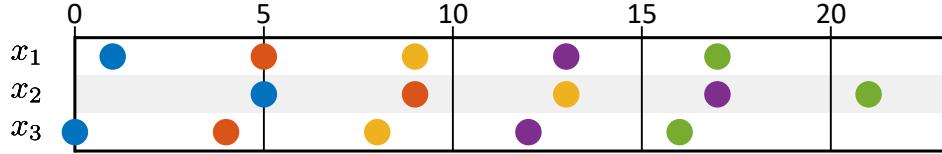


Figure C.23: Periodic sequence of token time stamps.

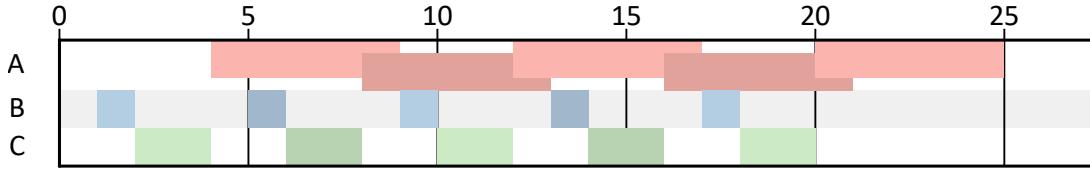


Figure C.24: Periodic schedule.

We can read the equation as follows. *Given a square matrix \mathbf{A} , do there exist combinations of a scalar $\lambda \in \mathbb{T}$ and a vector \mathbf{x} , for which Equation C.2 holds?*

If scalar λ and vector \mathbf{x} are a solution for the eigenvalue equation for a matrix \mathbf{A} , then we call λ an *eigenvalue* of \mathbf{A} and we call \mathbf{x} an *eigenvector* of \mathbf{A} . We can easily check that if vector \mathbf{x} is an eigenvector of \mathbf{A} , then also vector $c \otimes \mathbf{x}$ is an eigenvector of \mathbf{A} for any $c \in \mathbb{T}$ such that $c \neq -\infty$.

$$\mathbf{A}(c \otimes \mathbf{x}) = c \otimes (\mathbf{A}\mathbf{x}) = c \otimes (\lambda \otimes \mathbf{x}) = \lambda \otimes (c \otimes \mathbf{x})$$

Because of this fact, we usually select the eigenvector \mathbf{x} with norm $|\mathbf{x}| = 0$ as a representative of this class of solutions. Study of the eigenvalue equation is often called *spectral analysis*.

Definition C.30 (Eigenvalue, eigenvector). Let $\mathbf{A} \in \mathbb{T}^{n \times n}$. λ is an *eigenvalue* of \mathbf{A} and $\mathbf{x} \in \mathbb{T}^n$ an *eigenvector* of \mathbf{A} if $|\mathbf{x}| > -\infty$ and

$$\mathbf{A}\mathbf{x} = \lambda \otimes \mathbf{x}$$

Observe that the eigenvalue equation only applies to square matrices, since the size of the vector before and after multiplication with the matrix \mathbf{A} should be identical. Let matrix \mathbf{A} , vector \mathbf{x} and scalar λ be such that the equation holds. Then, the vectors $\mathbf{A}\mathbf{x}$ and \mathbf{x} are similar, each pair of corresponding elements of the vector differ exactly by the amount of λ .

Example C.16 (Dataflow Graph and Eigenvalue Equation). Consider the matrix \mathbf{G} that we derived in Example C.13 for the dataflow graph in Figure C.20. Note that \mathbf{G} is square and may have an eigenvector. Assume that the vector \mathbf{x} is an eigenvector of \mathbf{G} . This means that if we assume that the initial tokens in the dataflow graph are initially available in accordance with the time stamps in the vector \mathbf{x} , then by conclusion of an iteration of the dataflow graph (i.e., one firing from each of its actors), then the tokens in the graph are reproduced according to the time stamps in the vector $\mathbf{G}\mathbf{x} = \lambda \otimes \mathbf{x}$. In other words, all tokens in the graph are reproduced

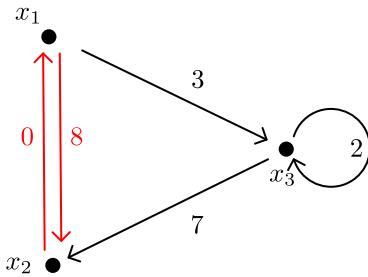


Figure C.25: Precedence Graph

exactly λ time units after they were initially available. This means that the continued execution of the following iterations of the dataflow graph can continue in a periodic fashion in which for every actor firing $k+1$ starts exactly λ time units after firing k . A solution to the eigenvalue equation therefore gives us a periodic schedule for the dataflow graph with a known throughput $1/\lambda$ (for any actor). The following equation is a solution to the eigenvalue equation for the matrix \mathbf{G} .

$$\begin{bmatrix} -\infty & 0 & -\infty \\ 8 & -\infty & 7 \\ 3 & -\infty & 2 \end{bmatrix} \begin{bmatrix} -4 \\ 0 \\ -5 \end{bmatrix} = 4 \otimes \begin{bmatrix} -4 \\ 0 \\ -5 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ -1 \end{bmatrix}$$

Note that the eigenvector is normal and therefore does not have positive time stamps. The eigenvalue of \mathbf{G} is $\lambda = 4$. \mathbf{G} does not have any other eigenvalues.

The eigenvalue and eigenvector give rise to a periodic schedule with period $\lambda = 4$. For example, if we assume as the initial time stamps of the initial tokens the eigenvector $[1 \ 5 \ 0]^T$, then we obtain the following periodic schedule.

$$\begin{cases} \sigma(A, k) = 4 + 4k \\ \sigma(B, k) = 1 + 4k \\ \sigma(C, k) = 2 + 4k \end{cases}$$

Figure C.23 shows a sequence of eigenvectors that represent the evolution of the initial-token time stamps of the dataflow graph of Figure C.20 according to this schedule. Figure C.24 shows the Gantt chart of the periodic execution of the dataflow graph of Figure C.20 according to the schedule σ . Note that it is indeed periodic.

Example C.17 (Railroad Network, Spectral Analysis). The eigenvalue equation

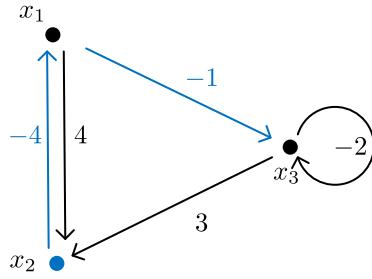


Figure C.26: Precedence Graph Eigenvector

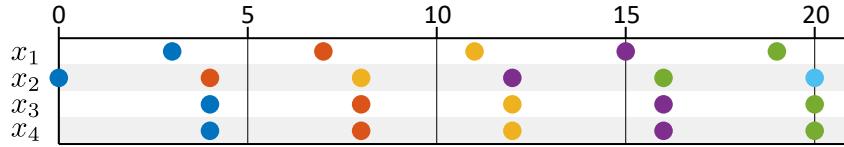


Figure C.27: Sequence of time-stamp eigenvectors of the self-timed execution of the railroad network

for the railroad network dataflow graph of Figure C.7 admits the following solution.

$$\begin{bmatrix} -\infty & 1 & 3 & -\infty \\ 1 & -\infty & -\infty & -\infty \\ 4 & -\infty & -\infty & 4 \\ 4 & -\infty & -\infty & 4 \end{bmatrix} \begin{bmatrix} -1 \\ -4 \\ 0 \\ 0 \end{bmatrix} = 4 \otimes \begin{bmatrix} -1 \\ -4 \\ 0 \\ 0 \end{bmatrix}$$

Assuming that we schedule the initial trains to depart according to the vector $\mathbf{x}(0) = [3 \ 0 \ 4 \ 4]^T$, the trains, under self-timed execution, will assume a periodic schedule.

$$\mathbf{x}(0) = \begin{bmatrix} 3 \\ 0 \\ 4 \\ 4 \end{bmatrix}, \mathbf{x}(1) = \begin{bmatrix} 7 \\ 4 \\ 8 \\ 8 \end{bmatrix} = 4 \otimes \mathbf{x}(0), \mathbf{x}(2) = \begin{bmatrix} 11 \\ 8 \\ 12 \\ 12 \end{bmatrix} = 8 \otimes \mathbf{x}(0), \mathbf{x}(3) = \begin{bmatrix} 15 \\ 12 \\ 16 \\ 16 \end{bmatrix} = 12 \otimes \mathbf{x}(0).$$

Figure C.27 shows the corresponding periodic sequence of time-stamp eigenvectors with a period equal to the eigenvalue $\lambda = 4$.

The *largest eigenvalue* of a matrix is typically determined by conversion of the matrix to a *precedence graph*.

Definition C.31 (Precedence-Graph Construction). Given a square matrix $\mathbf{A} \in \mathbb{T}^{n \times n}$, the *precedence graph* of \mathbf{A} is the weighted directed graph with nodes

$$V = \{x_1, \dots, x_n\}$$

The set of weighted edges is defined as follows.

$$E = \{(x_j, \mathbf{A}[i, j], x_i) \mid \mathbf{A}[i, j] \neq -\infty\}$$

Definition C.32 (Determining the Largest Eigenvalue). Let (V, E) be the precedence graph of the matrix \mathbf{A} . The largest eigenvalue of \mathbf{A} is equal to the *maximum cycle mean (MCM)* of the precedence graph. The cycle mean of a cycle is equal to the sum of the weights on all edges of the cycle divided by the number of edges in the cycle. A cycle in the precedence graph with the maximum cycle mean is called a *critical cycle* of the graph.

Definition C.33 (Determining the Eigenvector for the Largest Eigenvalue). Let (V, E) be the precedence graph of the matrix \mathbf{A} and let λ be the largest eigenvalue of \mathbf{A} and the MCM of the graph. Let x_m be a node on a critical cycle. Construct the weighted directed graph with nodes

$$V = \{x_1, \dots, x_n\}$$

and the set of weighted edges as follows.

$$E = \{(x_j, \mathbf{A}[i, j] - \lambda, x_i) \mid \mathbf{A}[i, j] \neq -\infty\}$$

Then an eigenvector \mathbf{x} is constructed by taking for $\mathbf{x}[k]$ the length of the longest path in the graph from x_m to x_k . If there is no path from x_m to x_k then $\mathbf{x}[k] = -\infty$. Note that the weighted directed graph is almost identical to the precedence graph, except that all edge weights have been lowered by λ . Note further that the resulting eigenvector does not need to be normal ($|\mathbf{x}| \geq 0$). If a normalized eigenvector is desired the resulting vector can easily be normalized afterwards.

Example C.18 (Precedence Graph). The precedence graph of the matrix from Example C.16 is shown in Figure C.25. The MCM of the graph is found in the cycle (x_1, x_2, x_1) . It has a cycle mean of $8/2 = 4$, indeed, the largest eigenvalue of the matrix. The corresponding eigenvector can be determined from the weighted graph in Figure C.26. This is the same graph as the precedence graph, but all edge weights are lowered with the eigenvalue $\lambda = 4$. The eigenvector is determined by starting from an arbitrary node x_r on the critical cycle and determining the longest path in the graph from that node to each of the other nodes in the graph. Note that the graph cannot have any cycles with positive weight since λ is the MCM. Therefore the longest path is bounded and in particular does not follow any cycles. The eigenvector \mathbf{x} has as its element $\mathbf{x}[k]$, the length of the longest path from the root node x_r to node x_k if such a path exists and it has the value $-\infty$ if there is no path from x_r to x_k .

Figure C.26 shows the longest paths from root node x_2 in blue and the result is indeed

the eigenvector

$$\mathbf{x} = \begin{bmatrix} -4 \\ 0 \\ -5 \end{bmatrix}$$

- A matrix with a precedence graph that is strongly connected, in which there is a path from every node to every other node, has only one normal eigenvector and eigenvalue.
- Therefore, a single-rate dataflow graph, for which the corresponding max-plus matrix has such a precedence graph, has a unique periodic schedule (up to a global shift in time).
- A single-rate dataflow graph that is strongly connected (there is a path of channels from every actor to every other actor) has exactly one eigenvalue. It may still have multiple independent eigenvectors.

Matrices may have more than one eigenvalue and matrices may also have multiple independent eigenvectors for the same eigenvalue. The following matrix, for example, has two eigenvalues.

$$\mathbf{G} = \begin{bmatrix} 1 & -\infty \\ -\infty & 2 \end{bmatrix}$$

$$\mathbf{G} \begin{bmatrix} 0 \\ -\infty \end{bmatrix} = 1 \otimes \begin{bmatrix} 0 \\ -\infty \end{bmatrix}$$

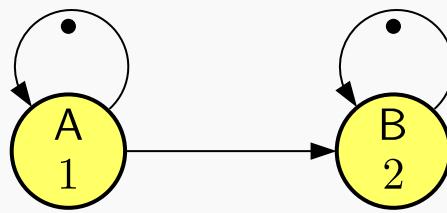
$$\mathbf{G} \begin{bmatrix} -\infty \\ 0 \end{bmatrix} = 2 \otimes \begin{bmatrix} -\infty \\ 0 \end{bmatrix}$$

We may gain some intuition from considering a dataflow graph that corresponds to this matrix, which is the following graph.



It has two actors without mutual dependencies, each proceeds at its own, different pace. This type of graph could also account for the situation of a single eigenvalue with multiple independent eigenvectors. Namely, if the firing duration of actor A of the above graph would have been 2, both independent vectors would still be eigenvectors, but with a single eigenvalue 2.

Another special case occurs with the matrix of a dataflow graph like the following.



Note that it is similar to the first graph, but now there is a dependency from the actor A to actor B. Imagine the self-timed execution of the graph. The self-timed schedule is identical to the self-timed schedule of the first graph. With the exception of its first firing, the dependencies of firings of B on firings of A are always satisfied by the time the dependency from the self-edge is satisfied.

The corresponding matrix is:

$$\mathbf{G} = \begin{bmatrix} 1 & -\infty \\ 3 & 2 \end{bmatrix}$$

Just like the first graph it has an eigenvalue 2 with the same eigenvector:

$$\mathbf{G} \begin{bmatrix} -\infty \\ 0 \end{bmatrix} = 2 \otimes \begin{bmatrix} -\infty \\ 0 \end{bmatrix}$$

1 is not an eigenvalue of this matrix due to the new dependency. There is, however, a vector, which is strictly speaking not an eigenvector, but has an eigenvector-like property. Consider the vector $\mathbf{x} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}$. It is easy to verify that:

$$\mathbf{G}^k \begin{bmatrix} -2 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 + k \\ 2k \end{bmatrix}$$

While for a regular eigenvector with eigenvalue λ all elements advance by an amount of λ every time the vector is multiplied by the matrix, in this case all elements of the vector also advance periodically, but with their own rate. In this case the first element advances by 1 and the second element advances by 2. We say that $\begin{bmatrix} -2 \\ 0 \end{bmatrix}$ is a *generalized eigenvector* with *generalized eigenvalue* $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$. A generalized eigenvalue is itself a vector that contains the periods for each of the elements. Such generalized eigenvectors are typical for dataflow graphs in which a fast subgraph produces tokens for a slower subgraph, but there are no dependencies going back.

Exercise C.17 (An image-based control system – eigenvalue, eigenvector).
Reconsider the IBC pipeline of Exercise C.3 and its max-plus matrix obtained in Exercise C.15. Since the graph is strongly connected, it has only one eigenvalue.

1. Compute the eigenvalue for the IBC pipeline.
2. Compute the corresponding (normal) eigenvector.
3. Provide the earliest non-negative initial-token time stamps that give a periodic self-timed schedule with the eigenvalue as its period. Also give that schedule.

See answer

Exercise C.18 (A producer-consumer pipeline – eigenvalue, eigenvector).

Reconsider the producer-consumer pipeline of Exercise C.2, with function f_2 for the filter actor F , and its max-plus matrix obtained in Exercise C.13. Since the graph is strongly connected, it has only one eigenvalue.

1. Compute the eigenvalue for the producer-consumer pipeline.
2. Compute the corresponding (normal) eigenvector.
3. Provide the earliest non-negative initial-token time stamps that give a periodic self-timed schedule with the eigenvalue as its period. Also give that schedule.

See answer

C.10 Max-Plus Linear Systems State-Space Equations

C.10.1 Definition

Max-plus-linear event systems are often described with linear-algebra equations. The system is assumed to have an internal state that can be captured as a vector $\mathbf{x} \in \mathbb{T}^n$ of n time stamps, for some $n \in \mathbb{N}$. The inputs to the event system are collected as a sequence of vectors $\mathbf{u}(k)$, where vector $\mathbf{u}(k)$ contains the k -th sample from each of the input event sequences. In a similar way, the outputs of the system are collected in a sequence $\mathbf{y}(k)$ of vectors. As the system operates, the next state, $\mathbf{x}(k+1)$ may depend on the current inputs $\mathbf{u}(k)$ and the current state, $\mathbf{x}(k)$. The outputs $\mathbf{y}(k)$ may also depend on the current state and the current inputs. Note that the current state and current input refers to the event index, not to the point in time. The various dependencies, being linear, can be characterized by matrices. Such a description of a max-plus-linear system is called a *state-space representation*.

Definition C.34 (State-Space Representation). A single-rate max-plus linear and index-invariant system with finite-dimensional state can be characterized using the following set of max-plus-linear matrix-vector equations.

$$\begin{cases} \mathbf{x}(k+1) = \mathbf{Ax}(k) \oplus \mathbf{Bu}(k) \\ \mathbf{y}(k) = \mathbf{Cx}(k) \oplus \mathbf{Du}(k) \end{cases}$$

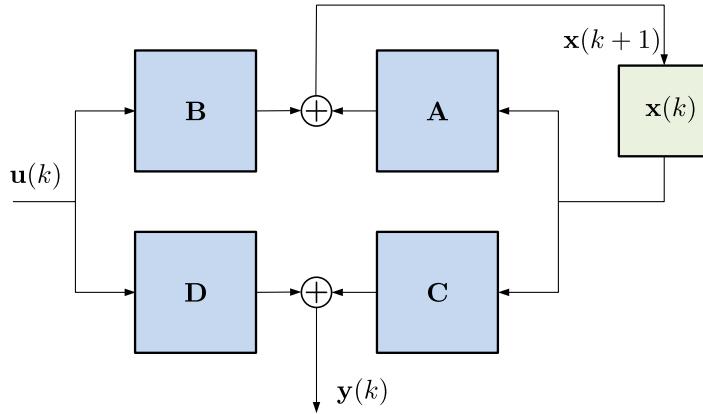


Figure C.28: State-Space Representation of a Max-Plus-Linear System.

The state-space representation is graphically represented in Figure C.28. Matrix **A** is called the *state matrix* (also called *system matrix*). **B** is the *input matrix*. **C** is the *output matrix* and **D** is the *feed-forward* matrix.

Example C.19 (Conveyor Belt, State-Space Representation). The conveyor belt of Example C.9 can be captured using the state-space representation. It requires only a single state variable, x , so its state vector has length $n = 1$. The state of the system represents all information from the past that is needed to describe its impact on the future. Let $x(k)$ be the time stamp at which the previous object has passed the entrance of the belt, i.e., the earliest time that the next object can be placed on the belt. With this definition we need to describe the four dependencies.

- the dependency from current state to next state describes the time between the moment that the entrance is free for object k and the time that it is free for object $k + 1$. This time is d/v , so matrix $\mathbf{A} = \begin{bmatrix} d/v \end{bmatrix}$.
- the dependency from the input $i(k)$ to the next state describes the time between the moment that object k arrives and the time that the entrance is free for object $k + 1$. This time is also d/v , so matrix $\mathbf{B} = \begin{bmatrix} d/v \end{bmatrix}$.
- the dependency from the current state to the output describes the time between the moment that the previous object has cleared the entrance and the time that the current object reaches the end of the belt. This is the duration of the object on the belt, i.e., l/v , $\mathbf{C} = \begin{bmatrix} l/v \end{bmatrix}$.
- the dependency from the arrival of object k to the time it reaches the end of the belt is also l/v , $\mathbf{D} = \begin{bmatrix} l/v \end{bmatrix}$.

The state-space equations for the conveyor belt are therefore as follows.

$$\begin{cases} x(k+1) = d/v \otimes x(k) \oplus d/v \otimes i(k) \\ o(k) = l/v \otimes x(k) \oplus l/v \otimes i(k) \end{cases}$$

The state-space equations are often collected into a single matrix-vector equation by collecting the produced vectors $\mathbf{x}(k+1)$ and $\mathbf{y}(k)$ into a single vector and similarly the input vectors $\mathbf{x}(k)$ and $\mathbf{u}(k)$ into a single vector. We obtain the equation below.

$$\begin{bmatrix} \mathbf{x}(k+1) \\ \mathbf{y}(k) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}(k) \\ \mathbf{u}(k) \end{bmatrix}$$

Verify that the result is indeed the same.

The state-space matrices define the dependencies of the next-state variables and the outputs on the current state variables and the inputs.

- We say that state variable $\mathbf{x}[k]$ depends on input $\mathbf{u}[m]$ if $\mathbf{B}[k, m] \neq -\infty$.
- We say that output $\mathbf{y}[k]$ depends on state variable $\mathbf{x}[m]$ if $\mathbf{C}[k, m] \neq -\infty$.
- We say that output $\mathbf{y}[k]$ depends on input $\mathbf{u}[m]$ if $\mathbf{D}[k, m] \neq -\infty$.

C.10.2 Dataflow Graphs as Max-Plus-Linear Systems

The state-space representation can also be used to describe any timed dataflow graph. Dataflow graphs have a natural definition of their internal state, namely the collection of the time stamps of all the tokens in the graph between iterations of the graph. Note that we have in fact already used such vectors in Section C.6.2.

The state-space matrices for a deadlock-free single-rate dataflow graph can be computed with a symbolic simulation method, very similar to Algorithm C.1. The method is presented in Algorithm C.2 below. The changes compared to Algorithm C.1 are indicated in red.

Algorithm C.2 (Deriving the State-Space Matrices with Symbolic Simulation). Perform the following steps.

- Introduce a time-stamp variable x_k , $1 \leq k \leq n$ for each of the n initial tokens in the dataflow graph **and a variable i_k , $1 \leq k \leq m$ for each of the m inputs in the graph.**
- Assign a symbolic time stamp $e_k = x_k$ to each of the initial tokens, represented by a symbolic time-stamp vector \mathbf{i}_k .
- **Assign a symbolic time stamp $e_{k+n} = i_k$ to each of the input tokens, represented by a symbolic time-stamp vector \mathbf{i}_{k+n} .**
- As long as there is an actor that is enabled and has not fired do the following.

- Select an arbitrary actor a that is enabled and has not fired yet in the symbolic simulation.
- Determine the symbolic time-stamp vector for the tokens that a produces after consuming tokens with symbolic time-stamp vectors $\mathbf{t}_1, \dots, \mathbf{t}_l$, from other actors and from inputs, in the same way as in Algorithm C.1:

$$\mathbf{t}_p = e(a) \otimes \bigoplus_{1 \leq i \leq l} \mathbf{t}_i$$

If actor a produces any outputs, record the symbolic time-stamp vectors of the outputs.

- Create the matrix

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

for the dataflow graph by collecting the symbolic time-stamp vectors for all the tokens, $\mathbf{t}_1, \dots, \mathbf{t}_n$, in the graph (they have returned to their original positions) in the order in which the variables x_1 to x_n were allocated and set row number $1 \leq k \leq n$ of the matrix equal to \mathbf{t}_k^T . (These rows form matrices \mathbf{A} and \mathbf{B} .) Set rows $n + 1 \leq k \leq n + m$ equal to the symbolic time stamps of the outputs that are produced. (These rows form matrices \mathbf{C} and \mathbf{D}).

Example C.20 (Railroad Network, Symbolic Simulation, Cont'd). The state vector consists of the time stamps of the tokens, $\mathbf{x} = [T_1, T_2, T_3, T_4]^T$.

We perform a symbolic simulation as in Example C.14, but with the generalizations of Algorithm C.2 to obtain the state-space equations.

For the symbolic time stamps, we include the input variable l representing the arrival $l(k)$ of a train on the input. We also assign the input a symbolic time stamp vector, which becomes \mathbf{i}_5 . (Recall that $\mathbf{i}_1, \dots, \mathbf{i}_4$ represent the four initial tokens.)

During the symbolic simulation we follow the same sequence of actor firings as in Example C.14. The symbolic time-stamp vectors for the actor firing starts and completions are now of length 5, including the input.

| actor | symbolic start | symbolic completion |
|-------|---|---|
| TEM | $[-\infty -\infty -\infty 0 -\infty]^T$ | $[-\infty -\infty -\infty 2 -\infty]^T$ |
| MST | $[-\infty -\infty -\infty 2 0]^T$ | $[-\infty -\infty -\infty 2 0]^T$ |
| TME | $[-\infty -\infty -\infty 2 0]^T$ | $[-\infty -\infty -\infty 4 2]^T$ |
| THA | $[-\infty 0 -\infty -\infty -\infty]^T$ | $[-\infty 1 -\infty -\infty -\infty]^T$ |
| TAH | $[0 -\infty -\infty -\infty -\infty]^T$ | $[1 -\infty -\infty -\infty -\infty]^T$ |
| THG | $[1 -\infty -\infty -\infty -\infty]^T$ | $[1 -\infty -\infty -\infty -\infty]^T$ |
| THE | $[1 -\infty -\infty -\infty -\infty]^T$ | $[4 -\infty -\infty -\infty -\infty]^T$ |
| EHV | $[4 -\infty -\infty 4 2]^T$ | $[4 -\infty -\infty 4 2]^T$ |

$$\begin{array}{ll}
 \text{TEA} & [-\infty \ -\infty \ 0 \ -\infty \ -\infty]^T \quad [-\infty \ -\infty \ 3 \ -\infty \ -\infty]^T \\
 \text{AMS} & [-\infty \ 1 \ 3 \ -\infty \ -\infty]^T \quad [-\infty \ 1 \ 3 \ -\infty \ -\infty]^T \\
 \text{TAS} & [-\infty \ 1 \ 3 \ -\infty \ -\infty]^T \quad [-\infty \ 2 \ 4 \ -\infty \ -\infty]^T
 \end{array}$$

Token T_1 is produced by actor **AMS**, T_2 by **THG**, T_3 and T_4 by **EHV**. Collecting the symbolic time-stamp vectors for the new tokens as the rows of the matrix gives the following result.

$$[\mathbf{A} \ \mathbf{B}] = \left[\begin{array}{cccc|c} -\infty & 1 & 3 & -\infty & -\infty \\ 1 & -\infty & -\infty & -\infty & -\infty \\ 4 & -\infty & -\infty & 4 & 2 \\ 4 & -\infty & -\infty & 4 & 2 \end{array} \right]$$

During the symbolic simulation we have also recorded the production of the output s , which coincides with the completion of the firing of actor **TAS**. This gives us the matrices **C** and **D**.

$$[\mathbf{C} \ \mathbf{D}] = \left[\begin{array}{ccc|c} -\infty & 2 & 4 & -\infty | -\infty \end{array} \right]$$

Note that because there is only one input, **B** and **D** have only one column, and because there is only one output, **C** and **D** have only one row. Thus the feed-forward matrix **D** contains only one element, $-\infty$, which indicates that there is no dependency of $s(k)$ on $l(k)$.

In the context of the state-space representation of a max-plus-linear system, we can determine the impulse response of a selected input k as follows. We create a sequence $\mathbf{u}(k)$ of input vectors such that the sequence $\mathbf{u}(m)[k]$ is the impulse sequence δ and $\mathbf{u}(m)[n] = \epsilon$ for all $n \neq k$. In terms of the sequence of input vectors, we have $\mathbf{u}(0) = \mathbf{i}_k$ and $\mathbf{u}(k) = \epsilon$ for all $k > 0$. (Recall that ϵ is the vector containing only $-\infty$ as its elements.) We then consider the corresponding sequence $\mathbf{y}(k)$ of output vectors, starting from the initial state $\mathbf{x}(0) = \epsilon$, as the impulse response. Very often we also want to observe the response of the impulse input on the sequence $\mathbf{x}(k)$ of state vectors. In that sense, the sequence $\mathbf{x}(k)$ is included when we talk about the impulse response of a certain input.

Just like dataflow graphs can be subject to self-timed scheduling, or can be executed according to a prescribed schedule (Definition C.2), we can also define a schedule for max-plus-linear systems in state-space representation. In this case, we prescribe the sequence of state vectors that are used for every iteration of the system. Such schedules can only be followed if they are valid, i.e., if all constraints, input constraints and internal constraints, are met.

Definition C.35 (Schedule of a Max-Plus-Linear System in State-Space Representation). A *schedule* σ of a max-plus-linear system in state-space representation with state matrix $\mathbf{A} \in \mathbb{T}^{n \times n}$ is a sequence of vectors from \mathbb{T}^n . The schedule is *valid* for a given input sequence $\mathbf{u}(k)$ if for all $k \in \mathbb{N}$,

$$\mathbf{A}\sigma(k) \oplus \mathbf{B}\mathbf{u}(k) \leq \sigma(k+1)$$

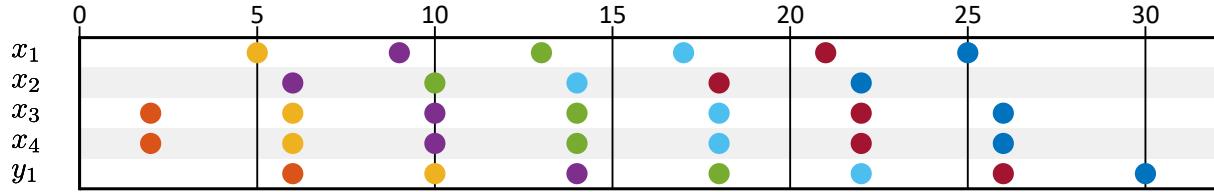


Figure C.29: Railroad network impulse response.

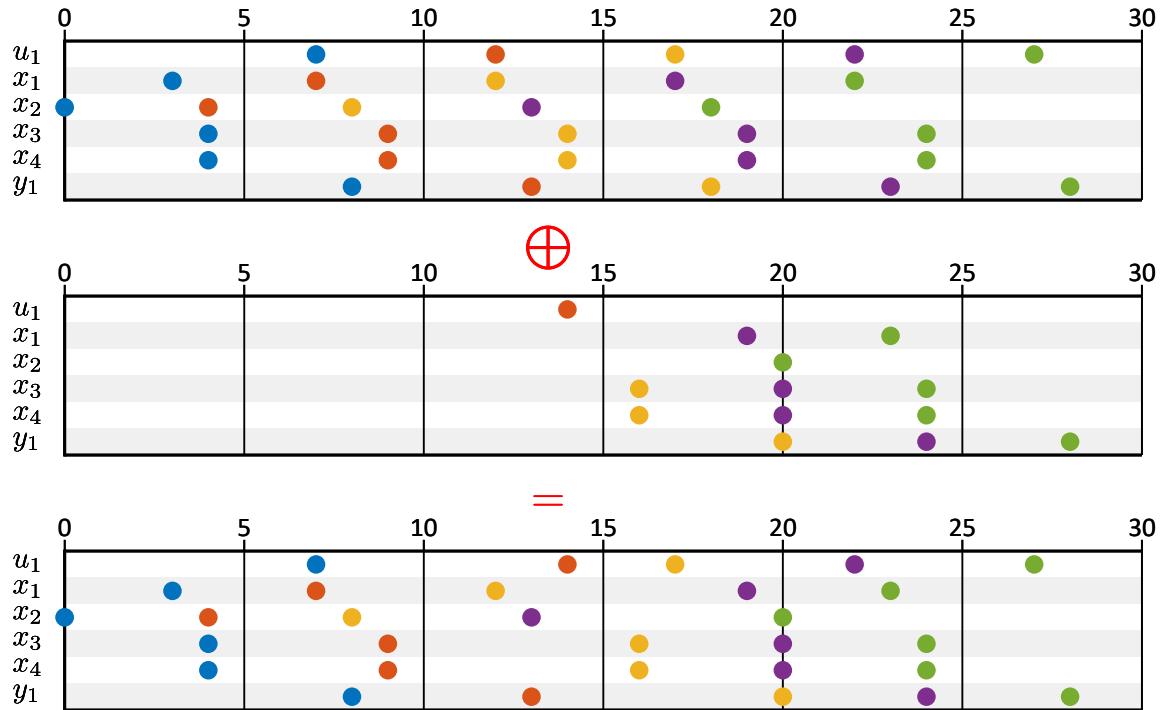


Figure C.30: Railroad network impulse response analysis.

In general, we describe the execution that follows from a given schedule (valid or not valid) as follows.

Definition C.36 (Execution According to a Schedule of a Max-Plus-Linear System in State-Space Representation). The sequence of system states and outputs of the execution of a max-plus-linear system with state-space matrices \mathbf{A} , \mathbf{B} , \mathbf{C} and \mathbf{D} with inputs $\mathbf{u}(k)$ and schedule $\sigma(k)$ is defined by the following equations.

$$\mathbf{x}(0) = \sigma(0)$$

$$\mathbf{x}(k+1) = \mathbf{Ax}(k) \oplus \mathbf{Bu}(k) \oplus \sigma(k+1)$$

$$\mathbf{y}(k) = \mathbf{Cx}(k) \oplus \mathbf{Du}(k)$$

For all $k \in \mathbb{N}$, $\sigma(k) \leq \mathbf{x}(k)$, and if the schedule is valid $\mathbf{x}(k) = \sigma(k)$

Example C.21 (Railroad-Network Impulse-Response Analysis in the State-Space Model). The impulse-response analysis discussed in Example C.12 can also be applied to investigate how disturbances in the operation impact the schedule or execution. For example, to study the impact of a late arriving input, or an extended duration of an operation.

Consider the periodic train schedule shown in the top trace in Figure C.30 in the form of a sequence of state vectors. Recall that from those state vectors and from the input sequence, all details of the trains operating in a self-timed fashion, can be reconstructed.

This particular periodic train schedule operates at a period of five hours, although a periodic schedule with a period of four hours also exists (see Figure C.27), i.e., it has a bit of slack. Let $\bar{\mathbf{p}}$ be the sequence of state vectors in the periodic schedule of Figure C.30. Note that the state vectors are a valid schedule (see Definition C.35) if we assume that all inputs arrive early enough, i.e., for all $k \in \mathbb{N}$,

$$\mathbf{A}\bar{\mathbf{p}}(k) \leq \bar{\mathbf{p}}(k+1)$$

In words, this means that all trains return, at the end of an iteration, in time to depart according to the next scheduled departure times. We assume that within an iteration of the schedule all trains depart as soon as possible.

Assume now that we want to study the impact on the schedule from a delay of two hours of the second train from Liège. I.e., it arrives at 14.00 hours instead of at 12.00 hours. We can use the impulse response on the state vectors, which is shown in Figure C.29. Let $\bar{\mathbf{h}}$ be the impulse-response vector sequence of Figure C.29. (Note that Figure C.29 additionally shows the impulse response on the output.)

$$\bar{\mathbf{h}} = \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ -\infty \end{bmatrix}, \begin{bmatrix} -\infty \\ -\infty \\ 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 5 \\ -\infty \\ 6 \\ 6 \end{bmatrix}, \begin{bmatrix} 9 \\ 6 \\ 10 \\ 10 \end{bmatrix}, \begin{bmatrix} 13 \\ 10 \\ 14 \\ 14 \end{bmatrix}, \dots$$

The impulse response, by definition, represents the impact of the first train arriving at time 0. It is easily adapted to the impact of the second train arriving at time 14 by an index delay of one sample and a time delay of 14, i.e., $14 \otimes \bar{\mathbf{h}}^1$, which is:

$$14 \otimes \bar{\mathbf{h}}^1 = \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ -\infty \end{bmatrix}, \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ -\infty \end{bmatrix}, \begin{bmatrix} -\infty \\ 16 \\ 16 \\ 16 \end{bmatrix}, \begin{bmatrix} 19 \\ -\infty \\ 20 \\ 20 \end{bmatrix}, \begin{bmatrix} 23 \\ 20 \\ 24 \\ 24 \end{bmatrix}, \begin{bmatrix} 27 \\ 24 \\ 28 \\ 28 \end{bmatrix}, \dots$$

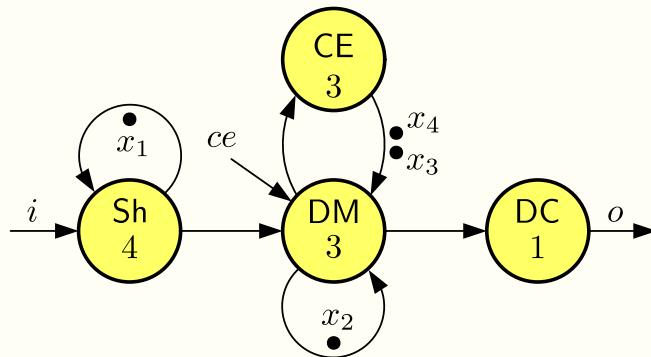
This sequence is shown in the second trace in Figure C.30.

The new train schedule can then be computed as the *superposition* (i.e., the maximum, \oplus) of the original schedule and the response to the delayed train:

$$\bar{\mathbf{p}} \oplus 14 \otimes \bar{\mathbf{h}}^1$$

The result is shown in the bottom trace in Figure C.30. We see that the third state vector is impacted by the delay, elements of the state are delayed by two time units. The fourth state vector is also impacted, but by one time unit only, and after that the schedule returns to the planned schedule \bar{p} . In terms of the outputs, the trains arriving at Schiphol, we see that the third train is delayed by two hours, the fourth train by one hour and subsequent trains are again on time.

Exercise C.19 (A wireless channel decoder – state-space model). Consider the following variant of the wireless channel decoder.



The model is a variant of the model of Exercise C.1, where, for simplicity, we omitted two self-edges. (You may check that the omitted self-edges did not impact the timing of the actor firings.) Because we want to investigate the effect of the availability of channel-estimation information on the timing of decoding output, we added a ce input to actor DM, in line with the construction explained after Theorem C.3.

1. Compute the state-space model of this dataflow graph using symbolic simulation.
2. Compute the impulse responses covering both states and output, up to four outputs, for impulses on i and ce .
3. Assume that symbols over input i arrive at times $4k$, for $k \geq 0$, and that inputs on ce arrive in time, so that they do not delay firings of Actor DM. Compute the first four outputs of the decoder assuming self-timed execution and availability of all initial tokens at time 0.
4. Assume now that the first execution of Actor CE takes 4 time units instead of 3. Use the model and the analysis done so far to analyze the impact of this longer execution time on the output times of the first four outputs.

See answer

C.11 Throughput Analysis

A max-plus-linear system may be limited in the number of outputs it can maximally produce per time unit. We call this the (*maximal*) *throughput* of the system. We can study throughput by considering the behavior of a max-plus-linear system for which an unlimited supply of inputs is present at time $-\infty$ and the first event kicks-off at time 0. The inputs are then not limiting the progress of the output production and any existing limitations are inherent in the system. Recall Definition C.15 that defines the throughput τ of an event sequence.

Definition C.37 (Maximal Throughput). The maximal throughput τ of a max-plus-linear system S on output k is equal to

$$\tau(S, k) = \tau(y_k)$$

where

$$\delta, \dots, \delta \xrightarrow{S} y_1, \dots, y_m$$

The throughput τ of a max-plus-linear system S is equal to

$$\tau(S) = \min_{1 \leq k \leq m} \tau(S, k)$$

If the system S has multiple outputs we consider the maximal throughput of S to be the lowest throughput among the outputs it produces.

Example C.22 (Maximal Throughput of the Conveyor Belt). Consider the conveyor belt of Example C.7. If we imagine an unlimited supply of boxes waiting to be transported on the belt we can easily compute that the output is equal to $a(k) = 5 + k$. Thus, the maximal throughput is computed as

$$\tau = \lim_{k \rightarrow \infty} \frac{k}{a(k)} = \lim_{k \rightarrow \infty} \frac{k}{5 + k} = 1$$

I.e., the maximal throughput of the conveyor belt is equal to 1 box per second.

The dataflow graph in Figure C.31 has outputs with different throughput. Output o_1 has throughput $1/5$, as determined by the dependencies between actors A and B. Output o_3 has throughput $1/6$, determined by the cycle through D and E. Finally, output o_2 depends on input from both B and E. The input from E is the slowest and the self-edge cycle on C and the firing duration of 2 of actor C make that C is fast enough to adopt the speed of the actor E. Therefore output o_2 has a throughput of $1/6$.

The maximal throughput of a system is determined by cyclic dependencies in the system and in terms of the state-space representation, between different elements of the state-vector of the system. Consider the precedence graph in Figure C.25. An edge in the graph from node x_i to node x_j , labelled with c embodies the constraint that $\mathbf{x}(k+1)[j] \geq c \otimes \mathbf{x}(k)[i]$. The edge from x_3 to x_2 implies that

$$\mathbf{x}(k+1)[2] \geq 7 \otimes \mathbf{x}(k)[3]$$

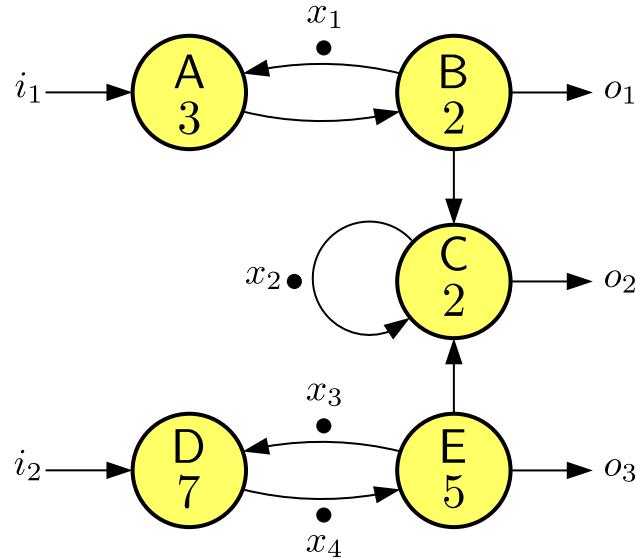


Figure C.31: Dataflow with outputs with different throughput

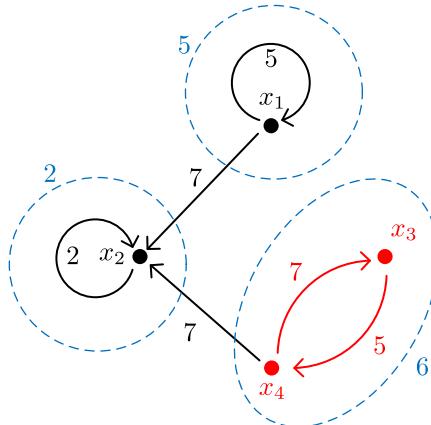


Figure C.32: Non-strongly-connected precedence graph

The critical cycle shown in red indicates that for all k

$$\mathbf{x}(k+1)[1] \geq \mathbf{x}(k)[2]$$

$$\mathbf{x}(k+1)[2] \geq 8 \otimes \mathbf{x}(k)[1]$$

and therefore

$$\mathbf{x}(k+2)[1] \geq 8 \otimes \mathbf{x}(k)[1]$$

and similar for $\mathbf{x}[2]$. Therefore, $\mathbf{x}(k)[1]$ advances by at least 8 from k to $k+2$, so on average at least 4 for every increment of k . This limits the throughput to at most $1/4$, and in general, the throughput for any state element is limited by the critical cycle, the cycle with the maximum cycle mean from which the state element is accessible.

The precedence graph in Figure C.32 corresponds to the matrix below, which, in turn, is

the state matrix of the dataflow graph of Figure C.31.

$$\begin{bmatrix} 5 & -\infty & -\infty & -\infty \\ 7 & 2 & -\infty & 7 \\ -\infty & -\infty & -\infty & 5 \\ -\infty & -\infty & 7 & -\infty \end{bmatrix}$$

The critical cycle is shown in red and has a cycle mean of 6. Clearly, x_3 and x_4 advance at an average rate of 6 time units per iteration. Also x_2 will advance with the same average rate, because there is a path from x_3 (and also from x_4) to x_2 . x_1 on the other hand, does not depend on x_3 or x_4 , there is no path from x_3 to x_1 . Therefore, x_1 will advance with an average rate of 5 time units per iteration. In general, every element of the state vector will advance, on average, with the rate of the cycle with the largest cycle mean from which it is accessible. In Figure C.32, the blue dashed lines show the strongly connected components of the precedence graph. All elements in a strongly connected component must have the same average rate. Every strongly connected component adopts the rate of the slowest strongly connected component from which it is accessible.

The throughput of an output of the system is determined by the slowest state element on which it depends. We can observe from the dataflow graph in Figure C.31 that output o_1 depends only on x_1 and therefore has a throughput of $1/5$. o_2 depends on the tokens, $\{x_1, x_2, x_3, x_4\}$. o_3 depends on $\{x_3, x_4\}$. Both outputs therefore adopt the average rate of x_3 and x_4 and have a throughput of $1/6$.

We can determine the dependencies from the state-space representation. The matrix below is the full state-space matrix of the same system. Recall that the matrix \mathbf{C} expresses the direct dependencies between the state vector and the outputs (the first row for o_1 , second row for o_2 , and third row for o_3). In general, an output y_k depends on state-vector element x_m if there is some state-vector element x_l such that $\mathbf{C}[k, l] \neq -\infty$ and x_l is accessible from x_m in the precedence graph of the state matrix \mathbf{A} . The afore-mentioned dependencies can now be verified using the state-space model and the precedence graph of Figure C.32.

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \left[\begin{array}{cccc|cc} 5 & -\infty & -\infty & -\infty & 5 & -\infty \\ 7 & 2 & -\infty & 7 & 7 & -\infty \\ -\infty & -\infty & -\infty & 5 & -\infty & -\infty \\ -\infty & -\infty & 7 & -\infty & -\infty & 7 \\ \hline 5 & -\infty & -\infty & -\infty & 5 & -\infty \\ 7 & 2 & -\infty & 7 & 7 & -\infty \\ -\infty & -\infty & -\infty & 5 & -\infty & -\infty \end{array} \right]$$

Theorem C.5 (Throughput). Let S be a max-plus-linear system with state matrix \mathbf{A} with largest eigenvalue λ . If for every element of the state vector there is an output that depends on it, then the maximal throughput of S is $1/\lambda$.

If \mathbf{A} does not have an eigenvalue, i.e., if its precedence graph has no cycles, then the throughput of S is not bounded.

Theorem C.6 (Throughput). Let S be a max-plus-linear system with state matrix \mathbf{A} with largest eigenvalue λ and let input sequences u_1, \dots, u_k have respective average rates of $1/\mu_1, \dots, 1/\mu_k$, then the average rate of the output sequence y is $\frac{1}{\lambda \oplus \mu_1 \oplus \dots \oplus \mu_k}$.

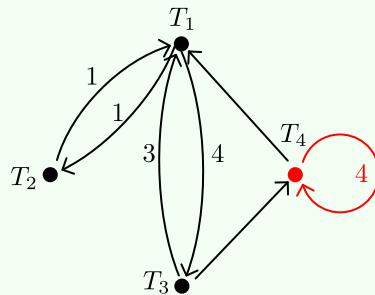
We have seen that the self-timed execution of a timed dataflow graph can be modelled as a max-plus-linear system using the state-space representation. Therefore, the above results allow us to predict the (maximal) throughput of a dataflow graph under self-timed execution.

Theorem C.7 (Self-timed Throughput). Let matrices \mathbf{A} , \mathbf{B} , \mathbf{C} and \mathbf{D} be the state-space representation of a timed dataflow graph G . Let \mathbf{A} have largest eigenvalue λ . Then the maximal throughput of G is equal to $1/\lambda$. If G is provided with input event sequences with rates $1/\mu_1, \dots, 1/\mu_k$, then the throughput of the self-timed execution is $\frac{1}{\lambda \oplus \mu_1 \oplus \dots \oplus \mu_k}$.

Example C.23 (Throughput of the Railroad Network). In Example C.20 we have seen that the state matrix of the railroad system is equal to

$$\mathbf{A} = \begin{bmatrix} -\infty & 1 & 3 & -\infty \\ 1 & -\infty & -\infty & -\infty \\ 4 & -\infty & -\infty & 4 \\ 4 & -\infty & -\infty & 4 \end{bmatrix}$$

The following is the corresponding precedence graph assuming the state vector $[T_1 \ T_2 \ T_3 \ T_4]^T$.



The maximum cycle mean is on the one-step cycle on state variable T_4 . All other state variables are accessible from T_4 , hence they will also adopt the rate of 4 time units per step. From the state matrices in Example C.20 we see that state variables T_3 and T_4 depend on input l , that output s depends on state variables T_2 and T_3 and that output s does not directly depend on input l .

Therefore, the throughput of the railroad network is $1/4$ and the throughput of the output sequence s will be the minimum of the throughput of the input sequence and the maximum throughput of the railroad network, which is $1/4$.

In general, the throughput of an output $\mathbf{y}[k]$ of a state-space model with given inputs is the minimum of the throughput of the event sequences of the state variables on which it

depends and of any inputs, $\mathbf{u}[m]$, on which it depends, either directly, according to matrix \mathbf{D} , or indirectly if any of the state variables on which it depends is accessible from any state variable that depends on $\mathbf{u}[m]$.

The output of a max-plus-linear system assumes the throughput of the input provided to the system, or the inherent throughput of the system itself, whichever is the smaller. That immediately tells us that if we make a pipeline of several systems, connecting the output of one system to an input of the next one, that the overall throughput will be the minimum among the throughput values of the individual systems.

Exercise C.20 (A wireless channel decoder – throughput). Reconsider the wireless channel decoder and its state-space model of Exercise C.19. Derive the maximal throughput that the decoder may achieve from its state-space model.

See answer

Exercise C.21 (An image-based control system – throughput). Reconsider the IBC pipeline of Exercise C.3. Derive the throughput that this pipeline may achieve from its state-space model.

See answer

Exercise C.22 (A producer-consumer pipeline – throughput). Reconsider the producer-consumer pipeline of Exercise C.7, with explicit input and output.

1. Derive the maximal throughput that this pipeline may achieve from its state-space model.
2. Assume now that the input of the pipeline is produced by the wireless channel decoder of Exercise C.20. What is the maximal achievable throughput of the combined system?

See answer

C.12 Latency Analysis

C.12.1 Definition

Latency is an important performance property that we can also compute from the state-space representation of a max-plus-linear system. The definition of latency for schedules of dataflow graphs has been defined as the maximum temporal distance among all $k \in \mathbb{N}$ between event k of an output and event k of an input. Here, we would like to define latency as an intrinsic property of the system, not just for a particular input sequence. This is not possible, however, without some assumptions about the inputs offered to the system.

We make the following assumptions on input. We will later see that we can relax those assumptions a bit. We assume that all inputs are provided with event sequences $\mathbf{u}[k]$ that

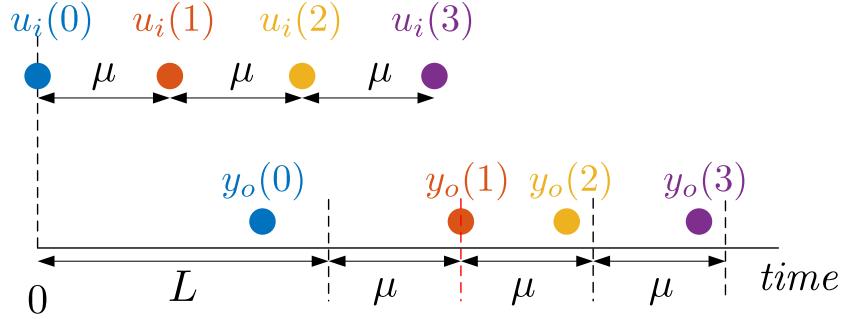


Figure C.33: Latency of a max-plus-linear system

are periodic with a period $\mu \geq \lambda$, where λ is the largest eigenvalue of the state matrix (i.e., the reciprocal of the maximal throughput of the system). If we offer inputs faster than λ , then the system will not be able to keep up, the outputs will be produced with a throughput of $1/\lambda$ and the latency will be unbounded. μ is a parameter of the analysis and the latency result may depend on it. The relation is monotone, however, i.e., a larger period μ cannot lead to a higher latency.

Definition C.38 (Latency of a Max-Plus-Linear System). The latency between input i and output o of a max-plus-linear system S for input period μ is

$$L(S, \mu, i, o) = \bigoplus_{k \in \mathbb{N}} y_o(k) - u_i(k)$$

where u_i is the μ -periodic event sequence (as defined in Definition C.14) and

$$\underbrace{\epsilon, \dots, \epsilon}_{i-1}, u_i, \epsilon, \dots, \epsilon \xrightarrow{S} y_1, \dots, y_m$$

Since u_i is μ -periodic, we have

$$L(S, \mu, i, o) = L(y_o, \mu)$$

where $L(s, \mu)$, the μ -periodic latency of an event sequence s , is defined as

$$L(s, \mu) = \bigoplus_{k \in \mathbb{N}} s(k) - \mu \cdot k$$

this is generalized to a sequence \bar{v} of vectors as

$$L(\bar{v}, \mu) = \mathbf{l}$$

where

$$\mathbf{l}[k] = L(\bar{v}[k], \mu)$$

I.e., latency is determined for each of the vector elements separately.

Figure C.33 illustrates the definition of latency. The top row shows the μ -periodic input sequence. The bottom row shows the corresponding output event sequence. In this example, output event $y_o(1)$ exhibits the maximum latency. The figure also illustrates another interpretation of the latency of system S . If the latency is L then the output sequence is never later than the μ -periodic sequence, scaled (i.e., delayed in time) by L . This scaled periodic sequence corresponds to the dashed vertical lines in the figure. The actual outputs are never later.

Example C.24 (Latency of the Manufacturing System). We study the latency of the manufacturing system of Example C.2 w.r.t. a desired period $\mu = 7$. That is, we would like to compute $L(out, 7)$. Since $\lambda = 5$ (you may wish to check this), we expect it to have a finite latency. The overall latency of the output sequence can be attributed to the combined effect of the input sequences and the elements of the initial state. We can therefore determine the latency from those four contributions separately using the superposition property. We assume that the system is operated from the initial state $\mathbf{x}(0) = [0 \ 0]^T$. Figure C.34 shows the Gantt charts for each of the four contributing parts separately. The first graph shows the latency due to the in_{bottom} input. It is the Gantt chart with input in_{bottom} equal to the μ -periodic input event sequence and $in_{top} = \epsilon$ and with initial state $\mathbf{x}(0) = \epsilon$. The latency due to the in_{bottom} input is equal to 10 since all output events occur at time $10 + k \cdot \mu$. The second Gantt chart, similarly, shows the latency due to the in_{top} input. The latency can be seen to be 16. The third Gantt chart investigates the latency due to the first state element, x_1 (the initial token on the edge from actor Pass1 to actor Int in the dataflow graph of Figure C.4). Both input sequences are set equal to ϵ and the initial state is set to $\mathbf{x}(0) = [0 \ -\infty]^T$. The latency is 16. The maximum value of 16 for $out(k) - k \cdot \mu$ occurs for the first output token (and decreases by 2 for every subsequent output token). The fourth graph shows that the latency due to the second element, x_2 of the state vector is 9. It is again the first output event that exhibits the largest delay. The Gantt chart starts from the initial state $\mathbf{x}(0) = [-\infty \ 0]^T$. Note that since many of the actor firings do not depend on the second state element (the initial token on the edge Rot—PaP), those actor firings all happen at time stamp $-\infty$ and are therefore not visible in the chart. The overall latency of the output is therefore equal to the maximum of the four components, $L(out, 7) = 16$.

Latency has been defined, and is analyzed in Example C.24, for strictly periodic input sequences with a fixed period μ . We can however exploit the properties of monotonicity and linearity to draw conclusions about the latency of a max-plus-linear and index-invariant system in response to inputs with other, non-periodic patterns too.

- Variable and bounded inter-arrival times. If an input event sequence has variable time distance between event arrivals, but a minimum time μ between any two

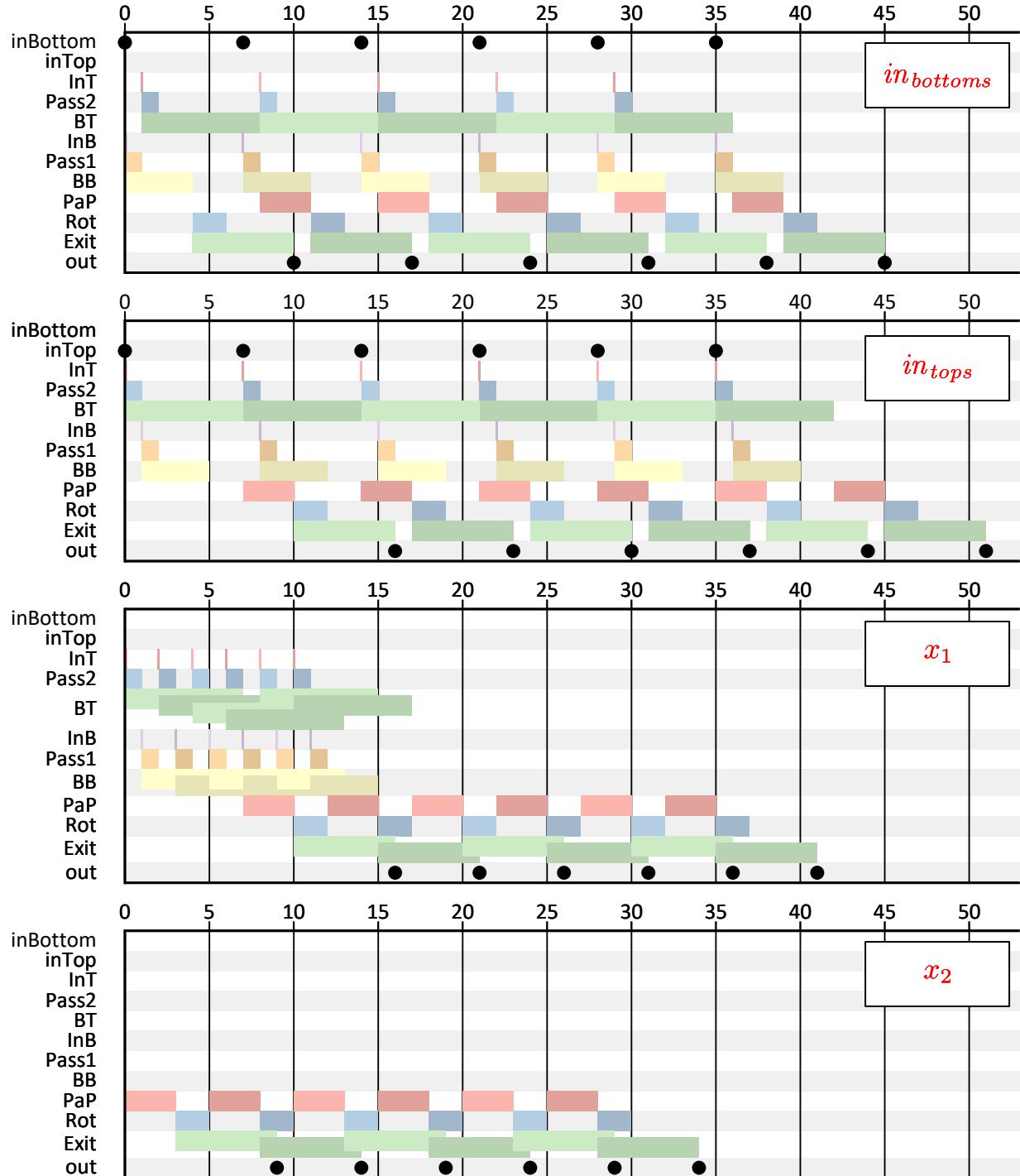


Figure C.34: Latency components manufacturing system

events, i.e., for all $k \in \mathbb{N}$,

$$i(k+1) - i(k) \geq \mu$$

then the latency can be bounded with the latency L^μ of the system for a μ -

periodic input sequence. The argument goes as follows.

The distance between the input i with the μ -periodic input is non-decreasing. Let $\Delta(k) = i(k) - \mu \cdot k$. Then $\Delta(k+1) \geq \Delta(k)$ for all $k \geq 0$. We know that the output event k does not depend on any input events with indices larger than k . Therefore, the input i limited up to event k produces the same outputs up to event k . Let i' be the input sequence that includes the events of i up to and including event k and let i_μ be the μ -periodic event sequence up to and including event k . Then $i' \leq \Delta(k) \otimes i_\mu$. Thus, by monotonicity and linearity, $o(k) \leq \Delta(k) \otimes o_\mu(k)$, where o_μ is the response to i_μ . Thus, $o(k) - i(k) \leq (\Delta(k) + o_\mu(k)) - (i_\mu(k) + \Delta(k)) = o_\mu(k) - i_\mu(k) \leq L^\mu$.

- Bounded arrival bursts. In a similar way, we can argue that we can determine a bound on the latency of inputs that may arrive in short bursts in which input events temporarily arrive with *shorter* inter arrival times than μ . Note that this can clearly increase the latency, but we can bound that increase. We need to limit such bursts, because otherwise the latency cannot be bounded. Assume that the input events are such that for all $k, m \in \mathbb{N}$, $i(m) - i(k) \geq (m - k - K) \cdot \mu$, i.e., up to K events could arrive simultaneously. Then there exists an input sequence i' with inter-arrival times bounded by μ , i.e., conforming to the previous case, such that $i \leq i' \leq (K \cdot \mu) \otimes i$. It is constructed from i by delaying events that are less than μ from their predecessor event to be at exactly μ distance from their predecessor. Let o' be the output corresponding to i' , then, by monotonicity, $o \leq o' \leq (K \cdot \mu) \otimes o$. Then $o(k) - i(k) \leq o'(k) - (i'(k) - K \cdot \mu) \leq o'(k) - i'(k) + K \cdot \mu \leq L^\mu + K \cdot \mu$. Hence, $L \leq L^\mu + K \cdot \mu$.

These derivations of latency bounds on different arrival models are due to Moreira [20].

C.12.2 Computing Latency from the State-Space Representation

We use the state-space representation of a max-plus-linear system to compute its latency. We start with the observation that the latency L is influenced by the initial state of the system, as we have seen in Example C.24. A larger initial vector may lead to outputs being produced later when they depend on the state. This increases the latency.

Every output of the system has a latency of its own. The latency vector $L(\bar{\mathbf{y}}, \mu)$ is a vector with the size of the number of outputs and contains as its elements the latencies for each of the outputs. The latency vector can be computed as:

$$L(\bar{\mathbf{y}}, \mu) = \bigoplus_{k \in \mathbb{N}} \mathbf{y}(k) - k\mu \otimes \mathbf{0} = \bigoplus_{k \in \mathbb{N}} (-k\mu) \otimes \mathbf{y}(k).$$

Following max-plus-linear-algebra computations (some background details are given below), we can derive that the latency for a max-plus-linear system with state-space matrices

A, **B**, **C**, and **D**, initial state $\mathbf{x}(0)$ and μ -periodic inputs is computed as follows:

$$L(\bar{\mathbf{y}}, \mu) = \mathbf{C} (-\mu \otimes \mathbf{A})^* (\mathbf{x}(0) \oplus (-\mu \otimes \mathbf{B}\mathbf{0})) \oplus \mathbf{D}\mathbf{0}, \quad (\text{C.3})$$

where $(-\mu \otimes \mathbf{A})^*$ denotes the so-called $*$ -closure of the matrix $-\mu \otimes \mathbf{A}$. The $*$ -closure of a square matrix \mathbf{M} is defined as

$$\mathbf{M}^* = \bigoplus_{k=0}^{\infty} \mathbf{M}^k.$$

Note that this closure exists in the latency computation of Equation C.3 if and only if the graph has a latency, or, equivalently, if the largest eigenvalue \mathbf{A} is not larger than μ . It can be efficiently computed exactly.

The $*$ -closure of a square matrix $\mathbf{M} \in \mathbb{T}^{n \times n}$ of size n is defined as follows

$$\mathbf{M}^* = \bigoplus_{k=0}^{\infty} \mathbf{M}^k.$$

It cannot be computed directly following the equation, because it involves the maximum of an infinite number of matrices. One can show however that it is sufficient to compute the maximum of the first n powers of \mathbf{M} .

$$\mathbf{M}^* = \bigoplus_{k=0}^{\infty} \mathbf{M}^k = \bigoplus_{k=0}^{n-1} \mathbf{M}^k$$

It is usually computed using the corresponding precedence graph and the observation that the elements of the matrix \mathbf{M}^k correspond to the length of the longest path consisting of k steps between nodes of the precedence graph. Therefore the elements of the $*$ -closure matrix correspond to the longest paths with an arbitrary number of steps. This can be determined using an all-pairs longest-path algorithm on the precedence graph.

The $*$ -closure is monotone. I.e., let $\mathbf{A} \leq \mathbf{B}$ such that \mathbf{B}^* exists, then \mathbf{A}^* exists and $\mathbf{A}^* \leq \mathbf{B}^*$.

As we have seen in Example C.24, the latency of a particular output depends on a number of different things, each of the inputs that are needed to produce the output, and the initial state. The principle of superposition tells us however, that the impact of each of these ingredients can be determined completely independently. This is even true for each of the elements, independently, of the initial state vector. We show that therefore, in general, the latency of a system can be characterized by two matrices, namely one, $\Lambda_{IO}^\mu \in \mathbb{T}^{k \times n}$ that contains the latency of every combination of an input and an output, and one, $\Lambda_{state}^\mu \in \mathbb{T}^{k \times m}$ that contains the latency of every combination of a state vector element with every output, where k is the number of outputs (length of the output vector), n is the number of inputs and m is the length of the state vector.

From the two matrices Λ_{IO}^μ and Λ_{state}^μ , the latency can be computed. For example, the latency from input number k to output number n is

$$(\Lambda_{IO}^\mu \mathbf{i}_k)[n]$$

$\Lambda_{IO}^\mu \mathbf{i}_k$ gives all output latencies for input k as a vector, we then pick element n from the resulting vector.

All output latencies due to the initial state $\mathbf{x}(0)$ on output k are computed as

$$\Lambda_{state}^\mu \mathbf{x}(0)$$

Latency of a max-plus-linear system. Assume a max-plus-linear system with state-space matrices \mathbf{A} , \mathbf{B} , \mathbf{C} and \mathbf{D} . For a given period μ , let vector $\mathbf{l}_i = L(\bar{\mathbf{u}}, \mu)$ be a vector of latencies for the inputs $\bar{\mathbf{u}}$.

Then the latency of the outputs $\bar{\mathbf{y}}$ given initial state $\mathbf{x}(0)$ and input sequence $\bar{\mathbf{u}}$ for period μ is equal to

$$L(\bar{\mathbf{y}}, \mu) = \Lambda_{state}^\mu \mathbf{x}(0) \oplus \Lambda_{IO}^\mu \mathbf{l}_i$$

where

$$\Lambda_{state}^\mu = \mathbf{C} (-\mu \otimes \mathbf{A})^*$$

$$\Lambda_{IO}^\mu = \Lambda_{state}^\mu (-\mu \otimes \mathbf{B}) \oplus \mathbf{D}$$

Observe that this result conforms to Equation C.3 given above.

If we think back of Figure C.28 showing the structure of the state-space equations we can find an intuitive interpretation of the equations. The state latency depends on any paths from the state vector to the output. Such a path follows an arbitrary number of iterations through the state matrix \mathbf{A} and finally follows the output matrix \mathbf{C} . Because of the period μ it gains a delay μ on every iteration, hence, the $*$ -closure of the matrix $-\mu \otimes \mathbf{A}$ finally followed by a multiplication with \mathbf{C} . Similarly, the equation for the IO latency can be explained. A path from input to output either goes directly via the feed forward matrix \mathbf{D} , or it goes through the input matrix \mathbf{B} , then an arbitrary number of iterations on the state matrix, again gaining μ with every iteration and finally following the output matrix \mathbf{C} .

We show the derivation of the latency matrices of a max-plus-linear system. The following equation can be shown to hold for the state vector $\mathbf{x}(k)$ by induction on k using the state-space equation $\mathbf{x}(k+1) = \mathbf{Ax}(k) \oplus \mathbf{Bu}(k)$.

$$\mathbf{x}(k) = \mathbf{A}^k \mathbf{x}(0) \oplus \left(\bigoplus_{m=0}^{k-1} \mathbf{A}^m \mathbf{B} \mathbf{u}(k-m-1) \right)$$

Using this result, the output vector can be computed as

$$\begin{aligned}\mathbf{y}(k) &= \mathbf{Cx}(k) \oplus \mathbf{Du}(k) \\ &= \mathbf{C} \left(\mathbf{A}^k \mathbf{x}(0) \oplus \bigoplus_{m=0}^{k-1} \mathbf{A}^m \mathbf{B} \mathbf{u}(k-m-1) \right) \oplus \mathbf{Du}(k) .\end{aligned}$$

From this, the output-latency vector \mathbf{l} can be computed using the periodic inputs $\mathbf{u}(k) = k \cdot \mu \otimes \mathbf{l}_i$ and the definition of latency of a vector sequence.

$$\begin{aligned}\mathbf{l}_o &= \bigoplus_{k \in \mathbb{N}} \mathbf{y}(k) - (k \cdot \mu) \otimes \mathbf{0} = \bigoplus_{k \in \mathbb{N}} (-k\mu) \otimes \mathbf{y}(k) \\ &= \bigoplus_{k \in \mathbb{N}} (-k\mu) \otimes \mathbf{C} \left(\mathbf{A}^k \mathbf{x}(0) \oplus \bigoplus_{m=0}^{k-1} \mathbf{A}^m \mathbf{B} \mathbf{u}(k-m-1) \right) \oplus \mathbf{Du}(k) \\ &= \mathbf{C} \bigoplus_{k \in \mathbb{N}} (-k\mu) \otimes \mathbf{A}^k \mathbf{x}(0) \oplus \\ &\quad \bigoplus_{k \in \mathbb{N}} (-k\mu) \otimes \left(\mathbf{C} \bigoplus_{m=0}^{k-1} (\mathbf{A}^m \mathbf{B}((k-m-1) \cdot \mu \otimes \mathbf{l}_i)) \oplus \mathbf{D}(k \cdot \mu \otimes \mathbf{l}_i) \right) \\ &= \mathbf{C} \bigoplus_{k \in \mathbb{N}} (-\mu \otimes \mathbf{A})^k \mathbf{x}(0) \oplus \bigoplus_{k \in \mathbb{N}} \left(\mathbf{C} \bigoplus_{m=0}^{k-1} (\mathbf{A}^m \mathbf{B}((-m-1) \cdot \mu \otimes \mathbf{l}_i)) \oplus \mathbf{Dl}_i \right) \\ &= \mathbf{C} (-\mu \otimes \mathbf{A})^* \mathbf{x}(0) \oplus \mathbf{C} \bigoplus_{k \in \mathbb{N}} \left(\bigoplus_{m=0}^{k-1} (-\mu \otimes \mathbf{A})^m (-\mu) \otimes \mathbf{Bl}_i \right) \oplus \mathbf{Dl}_i \\ &= \mathbf{C} (-\mu \otimes \mathbf{A})^* \mathbf{x}(0) \oplus \mathbf{C} \left(\bigoplus_{m \in \mathbb{N}} (-\mu \otimes \mathbf{A})^m \right) (-\mu) \otimes \mathbf{Bl}_i \oplus \mathbf{Dl}_i \\ &= \mathbf{C} (-\mu \otimes \mathbf{A})^* \mathbf{x}(0) \oplus \mathbf{C} (-\mu \otimes \mathbf{A})^* (-\mu \otimes \mathbf{B}) \mathbf{l}_i \oplus \mathbf{Dl}_i \\ &= \mathbf{C} (-\mu \otimes \mathbf{A})^* \mathbf{x}(0) \oplus (\mathbf{C} (-\mu \otimes \mathbf{A})^* (-\mu \otimes \mathbf{B}) \oplus \mathbf{D}) \mathbf{l}_i\end{aligned}$$

If we split the result into a part that depends on the initial state and a part that depends on the input latency, we get

$$\mathbf{l}_o = \Lambda_{state}^\mu \mathbf{x}(0) \oplus \Lambda_{IO}^\mu \mathbf{l}_i$$

where

$$\Lambda_{state}^\mu = \mathbf{C} (-\mu \otimes \mathbf{A})^*$$

$$\Lambda_{IO}^\mu = \mathbf{C} (-\mu \otimes \mathbf{A})^* (-\mu \otimes \mathbf{B}) \oplus \mathbf{D} = \Lambda_{state}^\mu (-\mu \otimes \mathbf{B}) \oplus \mathbf{D}$$

Example C.25 (Latency of the Manufacturing System, Cont'd). We continue Example C.24. We have in the mean time seen how we can algebraically compute the latency result.

The state-space representation of the manufacturing system can be determined with the methods described in Section C.10 with the following outcome.

$$\mathbf{A} = \begin{bmatrix} 2 & -\infty \\ 12 & 5 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1 & 2 \\ 6 & 12 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 16 & 9 \end{bmatrix}, \mathbf{D} = \begin{bmatrix} 10 & 16 \end{bmatrix}$$

We consider latency for the desired period $\mu = 7$ as in Example C.24. We can now compute the matrices that characterize the latency of the manufacturing system.

$$-\mu \otimes \mathbf{A} = \begin{bmatrix} -5 & -\infty \\ 5 & -2 \end{bmatrix}$$

$$(-\mu \otimes \mathbf{A})^* = (-\mu \otimes \mathbf{A})^0 \oplus (-\mu \otimes \mathbf{A})^1 = \begin{bmatrix} 0 & -\infty \\ -\infty & 0 \end{bmatrix} \oplus \begin{bmatrix} -5 & -\infty \\ 5 & -2 \end{bmatrix} = \begin{bmatrix} 0 & -\infty \\ 5 & 0 \end{bmatrix}$$

Note that in the above computation we have computed the $*$ -closure using a result that is described in the background discussion on the $*$ -closure matrix.

$$\begin{aligned} \Lambda_{state}^\mu &= \mathbf{C} (-\mu \otimes \mathbf{A})^* \\ &= \begin{bmatrix} 16 & 9 \end{bmatrix} \begin{bmatrix} 0 & -\infty \\ 5 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 16 & 9 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \Lambda_{IO}^\mu &= \Lambda_{state}^\mu (-\mu \otimes \mathbf{B}) \oplus \mathbf{D} \\ &= \begin{bmatrix} 16 & 9 \end{bmatrix} \begin{bmatrix} -6 & -5 \\ -1 & 5 \end{bmatrix} \oplus \begin{bmatrix} 10 & 16 \end{bmatrix} = \begin{bmatrix} 10 & 14 \end{bmatrix} \oplus \begin{bmatrix} 10 & 16 \end{bmatrix} \\ &= \begin{bmatrix} 10 & 16 \end{bmatrix} \end{aligned}$$

Note that they indeed give us exactly the four components of the output latency that we could see from the Gantt charts in Figure C.34. The latency due to the in_{bottom} input is 10 (first element of Λ_{IO}^μ). The latency due to the in_{top} input is 16 (second element of Λ_{IO}^μ). The latency due to the two initial state elements x_1 and x_2 are 16 and 9 (the elements of Λ_{state}^μ), respectively.

From the two matrices, the output latency can be directly computed. Let the initial state be $\mathbf{x}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and assume that $\mu = 7$, that the latency of input sequence

in_{bottom} is 9 and the latency of input sequence in_{top} is 2, so that $\mathbf{l}_i = \begin{bmatrix} 9 \\ 2 \end{bmatrix}$. Then, the output latency can be computed as follows.

$$\begin{aligned} L(\bar{\mathbf{y}}, \mu) &= \Lambda_{state}^\mu \mathbf{x}(0) \oplus \Lambda_{IO}^\mu \mathbf{l}_i \\ &= \begin{bmatrix} 16 & 9 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 10 & 16 \end{bmatrix} \begin{bmatrix} 9 \\ 2 \end{bmatrix} \\ &= 16 \oplus 19 \\ &= 19 \end{aligned}$$

It follows directly from the monotonicity properties of max-plus algebra (Section C.8), monotonicity of the $*$ -closure and Equation C.3 that latency is monotone in:

- *the period μ* , larger μ does not increase latency;
- *the latency of the inputs*, smaller latency of the input sequences does not increase the latency of the output sequences;
- *the initial state*, a smaller initial state does not increase the latency of the output sequences;
- *any of the state-space matrices*, smaller matrices do not increase the latency of the output sequences.

Exercise C.23 (An image-based control system – latency). Reconsider the IBC pipeline of Exercise C.3. Derive the latency $L(o, 4)$ of the output from the state-space model of the IBC pipeline, assuming initial tokens are available at time 0.

See answer

Exercise C.24 (A wireless channel decoder – latency). Reconsider the wireless channel decoder and its state-space model of Exercise C.19. Derive the latency $L(o, 4)$ of the output from the state-space model of the WCD dataflow graph, assuming initial tokens are available at time 0 and assuming 4-periodic input sequences for i and ce .

See answer

Exercise C.25 (A producer-consumer pipeline – latency). Reconsider the producer-consumer pipeline of Exercise C.7, with explicit input and output.

1. Derive the latency $L(o, 2.5)$ of the output from the state-space model of the pipeline, assuming a 2.5-periodic input sequence and availability of initial tokens at time 0. (You may consider using the CMWB to compute the needed matrices.)



Figure C.35: Manufacturing System, *Modern Times*, by Charlie Chaplin, ©Roy Export SAS

2. Assume now that the input of the pipeline is produced by the wireless channel decoder of Exercise C.24. What is the $L(o, 4)$ of the combined system? Assume that the inputs to the channel decoder are both the 4-periodic event sequence and that all initial tokens are available at time 0.

See answer

C.13 Stability

Stability is the property of a system to be resilient to disturbance and return to its original state or behavior. For max-plus-linear systems, stability of their operation can be related to the maximum throughput that they can sustain. Intuitively, if the system operates at maximum throughput there is no room, no slack, to recover from any disturbance to its operation. Disturbances could be incidental late arrival of inputs, or incidental extension of the duration of an operation. As in the unstable manufacturing system shown in Figure C.35, the disturbance from the nominal schedule diverges and the operation fails.

A max-plus-linear system is called stable under infinite input sequences i_1, \dots, i_k if for any finite disturbance to an input sequence, only a finite number of output events change.

Definition C.39 (Stability of a max-plus-Linear System). Let $i \xrightarrow{S} o$ be the behavior of a max-plus-linear event system S with input i . S is called *stable under input i* if for any $c \in \mathbb{T}$, $k \in \mathbb{N}$, and $i \oplus c \otimes \delta^k \xrightarrow{S} o'$, there is some $m \in \mathbb{N}$ such that for all $n \in \mathbb{N}$, $n \geq m$, if $o(n)$ is defined, then $o'(n) = o(n)$.

The definition is straightforwardly generalized to systems with multiple inputs and outputs by requiring that a disturbance on any of the inputs influences only a finite number of output events on all outputs.

Note that if o is finite the system is trivially stable under input i .

Example C.26 (Stability of the Conveyor Belt). We consider the conveyor belt as a max-plus-linear event system as in Example C.9, $i \xrightarrow{CB} o$, with inputs $i = [0, 0, 2, 2, 4, 4, \dots]$ and corresponding outputs $o = [5, 6, 7, 8, 9, 10, \dots]$. This represents an *unstable* situation, for there is a finite disturbance on the input that will

disturb an infinite number of outputs. An example of such a finite disturbance is $4 \otimes \delta^2$, which, when added to i gives the input $i_d = [0, 0, 4, 2, 4, 4, \dots]$ with corresponding output $o_d = [5, 6, 9, 10, 11, 12, \dots]$ and $o_d(k) \neq o(k)$ for all $k \geq 2$.

A stable operation, on the other hand, of the conveyor belt is obtained by replacing the input with $i' = [0, 0, 3, 3, 6, 6, \dots]$. In this case, $o' = [5, 6, 8, 9, 11, 12, \dots]$. When the same disturbance is applied we get $i'_d = [0, 0, 4, 3, 6, 6, \dots]$ and $o'_d = [5, 6, 9, 10, 11, 12, \dots]$ and $o'_d(k) \neq o'(k)$ only for $k = 2, 3$. In this case, any finite disturbance influences only a finite number of outputs.

Operating the conveyor belt at the maximum throughput makes it unstable. Lowering the rate of inputs, as in the second case, makes it stable.

The following gives a sufficient condition for stability of a max-plus-linear event system.

Let S be a max-plus-linear and index-invariant system with maximal throughput $\tau(S)$. S is stable under inputs $i_1 \dots i_k$ if for all k

$$\tau(i_k) < \tau(S)$$

We can also apply the concept of stability to a schedule for a max-plus-linear system given in its state-space representation. A max-plus-linear system is called stable under schedule $\sigma(k)$ and inputs $\mathbf{u}(k)$ if for a finite disturbance to the $\mathbf{u}(k)$ only a finite number of changes happen to the execution \mathbf{x} under $\sigma(k)$, i.e., there are only a finite number of changes to the state vectors and to the output vectors.

Definition C.40 (Stability of a Schedule). Let S be a linear event system with n inputs in state-space representation. Let S be executed according to schedule $\sigma(k)$ resulting in the sequence $\mathbf{x}(k)$ of state vectors and the sequence $\mathbf{y}(k)$ of output vectors. $\sigma(k)$ is called *stable for S under input u(k)* if for any $\mathbf{c} \in \mathbb{T}^n$ and $m \in \mathbb{N}$, the execution with schedule $\sigma(k)$ and inputs \mathbf{u}' with $\mathbf{u}'(m) = \mathbf{u}(m) \oplus \mathbf{c}$ and $\mathbf{u}'(k) = \mathbf{u}(k)$ for $k \neq m$, leads to a sequence $\mathbf{x}'(k)$ of state vectors and a sequence $\mathbf{y}'(k)$ of output vectors, such that there is some $l \in \mathbb{N}$ such that for all $k \in \mathbb{N}$, $k \geq l$, $\mathbf{x}'(k) = \mathbf{x}(k)$ and $\mathbf{y}'(k) = \mathbf{y}(k)$.

Example C.27 (Stability of the Railroad Network). In Example C.23, we have seen that the fastest train schedule can repeat every 4 hours. The schedule is shown in Figure C.27. The schedule assumes that all trains arriving from Liège, the inputs to the system, are in time for the schedule to be valid.

Any schedule with a period smaller than four is *unstable* (and not valid, according to Definition C.35). The first sequence in Figure C.36 shows a schedule σ_3 with period three and the actual sequences according to the execution. Deviations between schedule and actual execution are shown in the figure by red open circles denoting the scheduled time stamps and the filled circles denoting the actual time stamps on those events that do not occur according to their original schedule. The figure shows that the schedule cannot be followed and the deviation from the schedule increases with

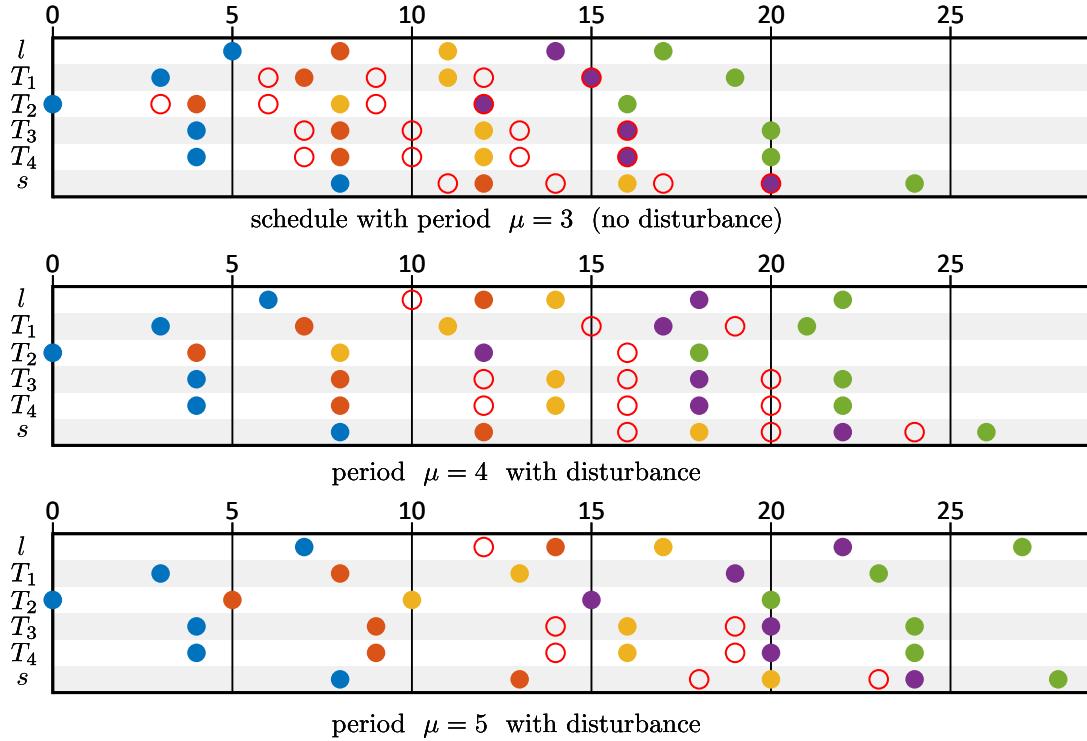


Figure C.36: Stability in the railroad network

subsequent iterations. We say that the schedule is *unstable*.

The second sequence shows execution according to a periodic schedule σ_4 with period four, i.e., equal to the largest eigenvalue of the state matrix. This situation is called *marginally stable* since disturbances that occur, neither dissolve nor diverge. It is unstable according to Definition C.40. A disturbance is introduced in the sequence by the second train from Liège arriving two hours late. As a consequence the execution deviates from the schedule (by two hours) and this deviation remains present in all future states and outputs.

Now assume that we introduce some slack in the schedule and we operate at a period of 5 hours instead of 4. This is shown in the third sequence with schedule σ_5 . Again, a disturbance is introduced by the second train being late by two hours. This time, the system is *stable*. The disturbance diminishes with subsequent iterations. For example, the third train to Schiphol is late by two hours, the fourth train is still late, but only by one hour and the following trains are on time again.

A valid schedule is stable if the throughput it requires from the system is lower than the maximal throughput the system can sustain.

Let S be a max-plus-linear system in state-space representation with state matrix \mathbf{A} . Let λ be the largest eigenvalue of \mathbf{A} . Let σ be a valid schedule for S under inputs $\mathbf{u}(k)$. Then σ is stable for S under inputs $\mathbf{u}(k)$ if the throughput of the event sequences for

each of the state elements in $\sigma(k)$ are smaller than $1/\lambda$.

Exercise C.26 (A wireless channel decoder – stability). Reconsider the wireless channel decoder and its state-space model of Exercise C.19.

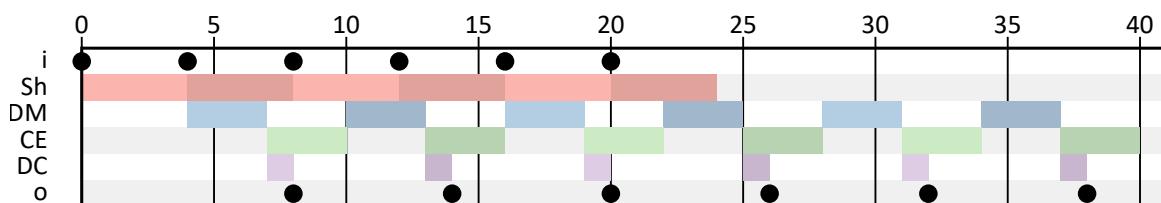
1. Consider the self-timed schedule, assuming initial tokens are available at time 0 and assuming the 4-periodic input sequence for i and the delayed 4-periodic input sequence for ce with the first input arriving at time 4. Is this schedule stable? Motivate your answer. If it is not stable, provide an example of a disturbance and its effect on the schedule that illustrates the instability.
2. Consider now the variant of the WCD model without input i . This allows us to investigate the impact of ce on the stability of the system. Note that the self-edge and execution time of Actor Sh ensure that input symbols are processed according to the self-timed schedule of the previous item. So the self-timed schedule, under the same assumptions on initial state and input ce , does not change with respect to the previous item. Is the resulting system stable for the assumed input on ce ? Motivate your answer. If it is not stable, provide an example of a disturbance and its effect on the schedule that illustrates the instability.

See answer

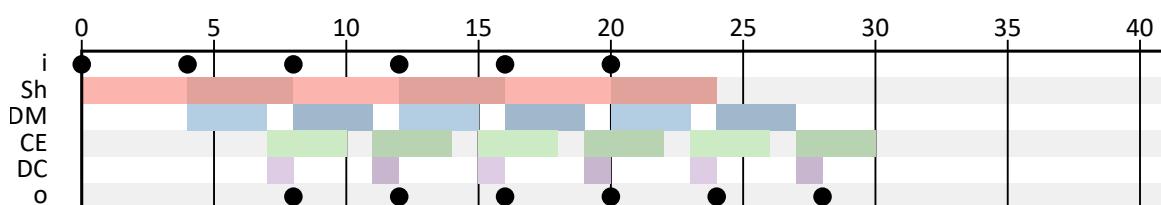
C.14 Answers to Exercises

Exercise C.1 (A wireless channel decoder – understanding dataflow).

1. If you entered the model correctly, the CMWB gives the following Gantt chart:



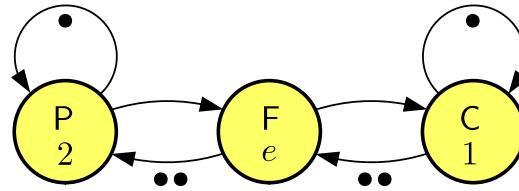
2. The CMWB gives the following Gantt chart:



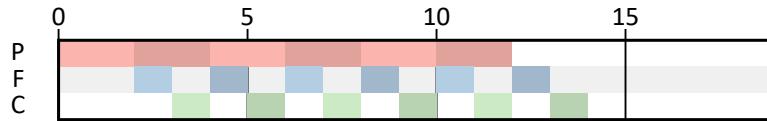
3. Given the input sequence and the duration of the **Sh** actor, the decoder cannot decode symbols faster than 1 symbol per 4 time units. With two tokens on the channel from CE to DM, it operates at this maximum rate; with only one token, it does not. So the most recent channel-estimation information that DM can use is two symbols old.

Exercise C.2 (A producer-consumer pipeline – creating dataflow models).

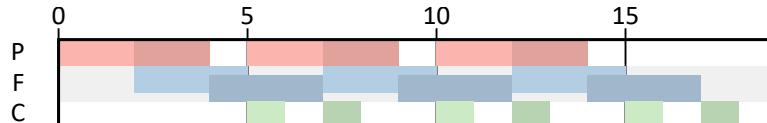
1. The pipeline can be modeled as follows, where $e \in \{1, 3\}$ is the execution time of the filtering actor F.



2. With $e = 1$, the CMWB gives the following Gantt chart:



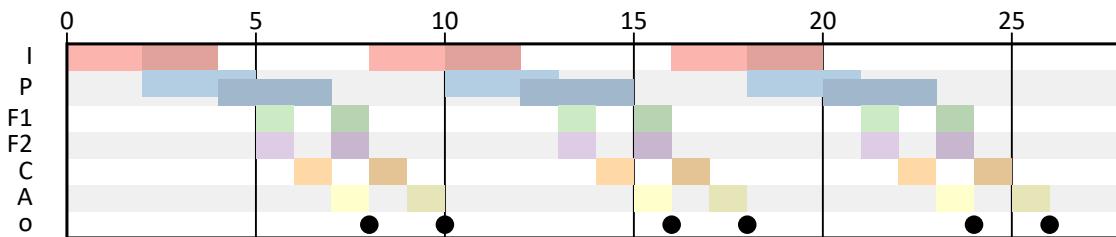
3. The CMWB gives the following Gantt chart when $e = 3$:



Note the overlap between firings of Actor F and the hick-ups in the production of samples due to a full buffer between P and F.

Exercise C.3 (An image-based control system – understanding and creating dataflow models).

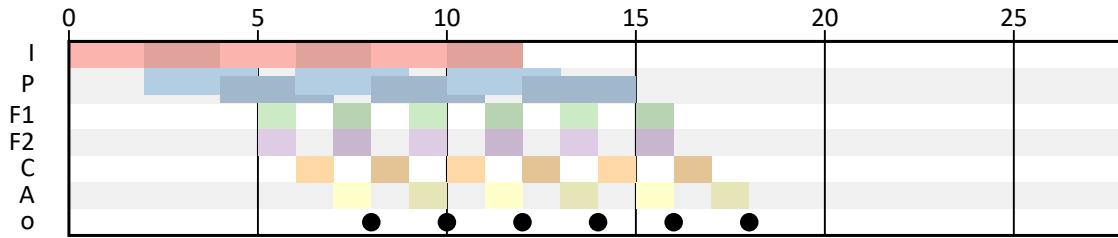
1. The CMWB gives the following Gantt chart:



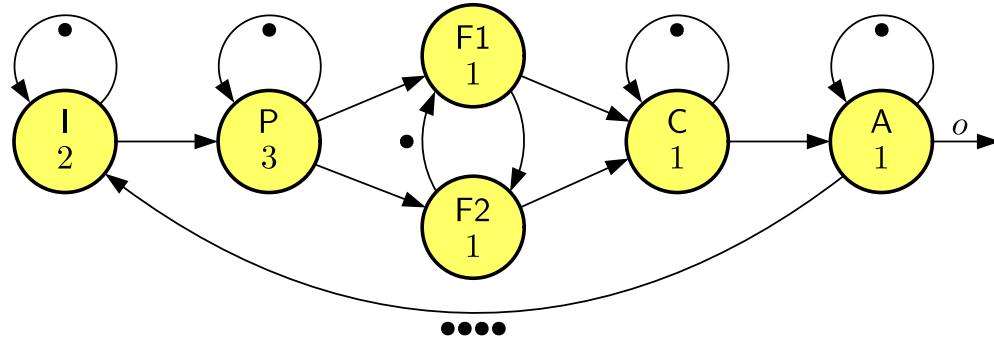
2. The IBC pipeline shows a repetitive pattern in which two outputs occur every eight time units. Hence, the average actuation rate is one actuation per four time units.

(Note that the actuation pattern is not strictly periodic, which may be undesirable from a quality-of-control perspective.)

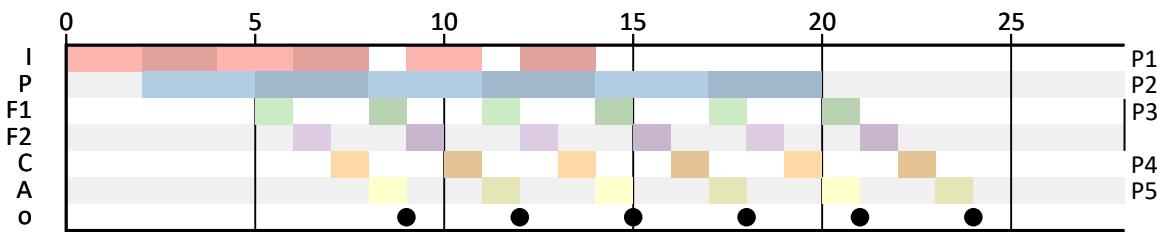
3. With a pipelining depth of four, the IBC pipeline operates at its maximum actuation rate of one actuation every two time units. The following Gantt chart can be obtained from the CMWB.



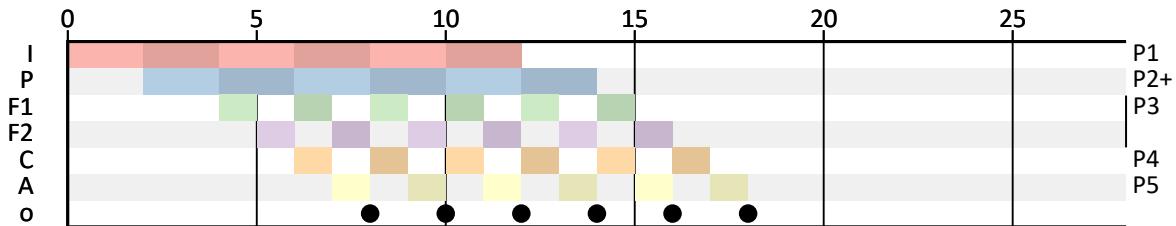
4. The fact that an actor is executing on a single processor can be captured in a dataflow model with a self-loop with one initial token. This prohibits actor executions to overlap. The mapping and scheduling of the two feature processing actors F1 and F2 can be modeled with two channels between those actors, one in each direction, with an initial token on the channel to F1. The latter ensures that F1 goes first among the two.



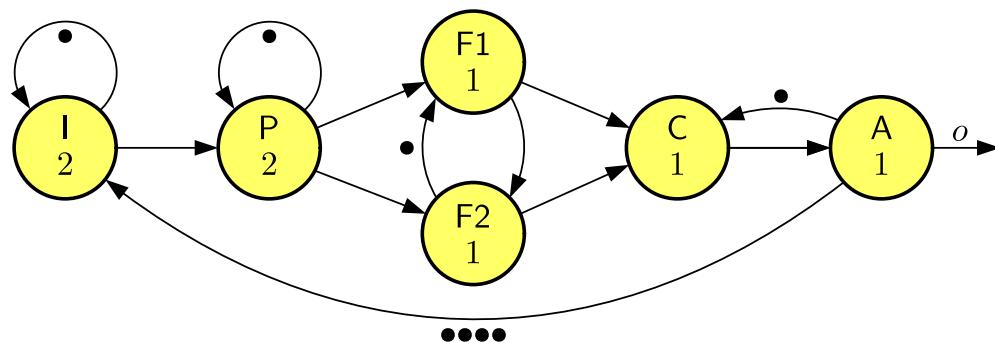
5. The following Gantt chart can be obtained from the CMWB. The processors P1 to P5 that execute the actors have been added at the right side.



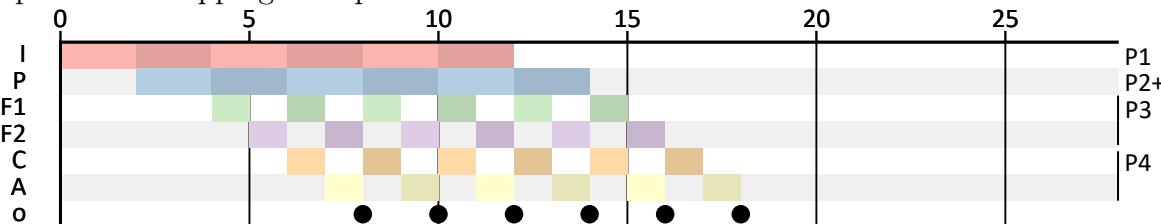
6. From the previous part, it is clear that the execution of Actor P is a bottleneck. Processor P2 executing P should be sped up by at least a factor 1.5 so that a P firing does not take more than two time units. The following Gantt chart illustrates the execution with the improved processor.



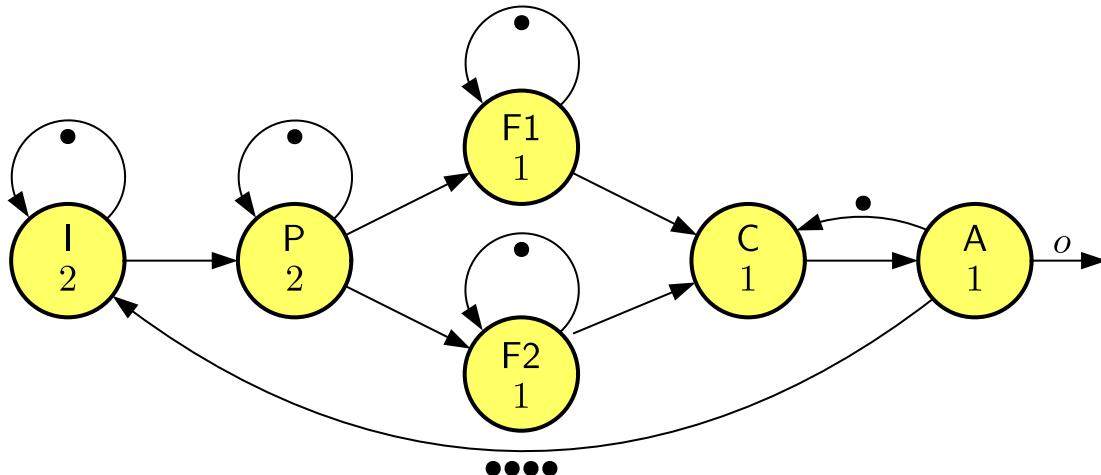
7. Actors C and A can be mapped onto a single processor, scheduling their firings alternatingly, starting with C. This reduces the number of processors to four. The dataflow model then becomes as follows:



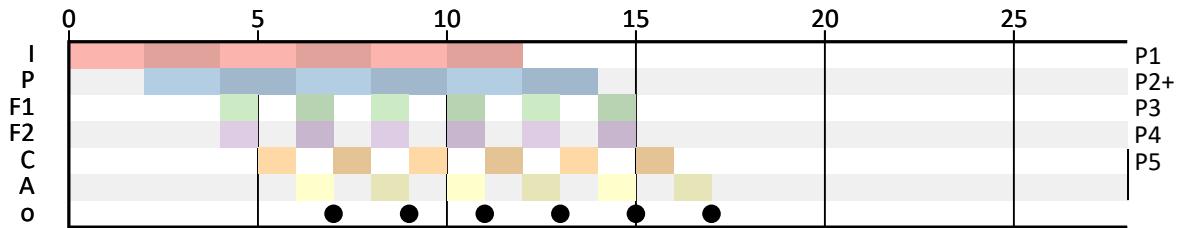
Note that the self-loops on C and A can be omitted. The Gantt chart does not change, except for the mapping onto processors.



8. The actuation delay of the IBCS of the previous item is 8. One option to improve it is to use faster processors. Another option is to give both feature processing actors their own processor:

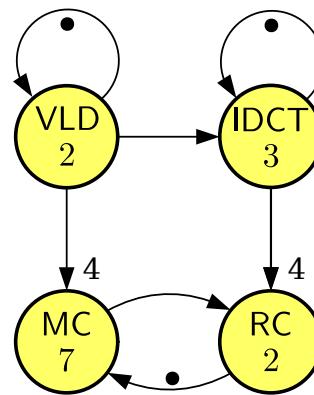


This leads to an actuation delay of 7:

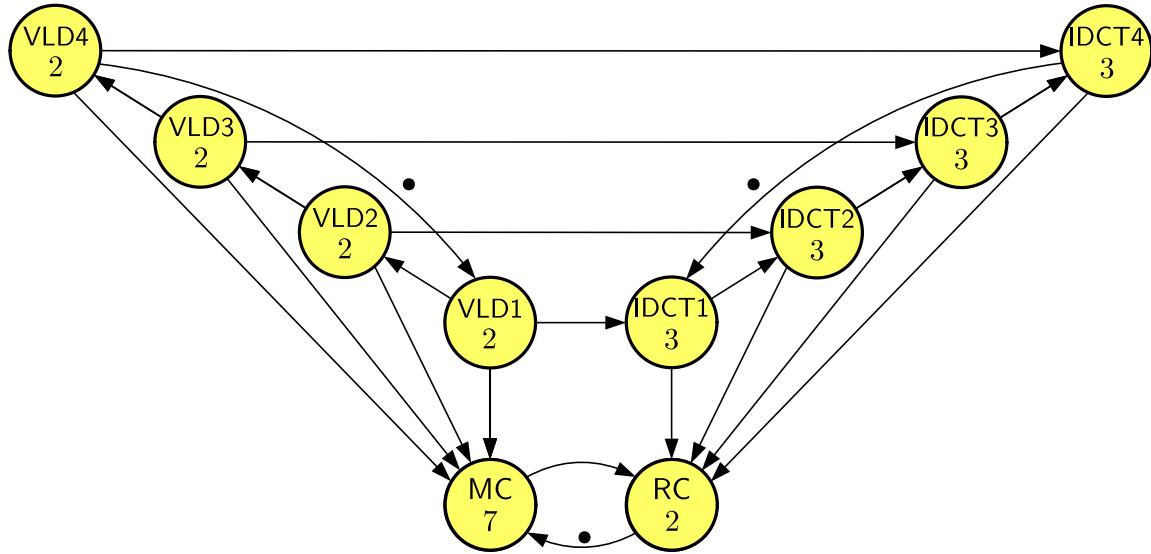


Exercise C.4 (A video decoder - multi-rate dataflow).

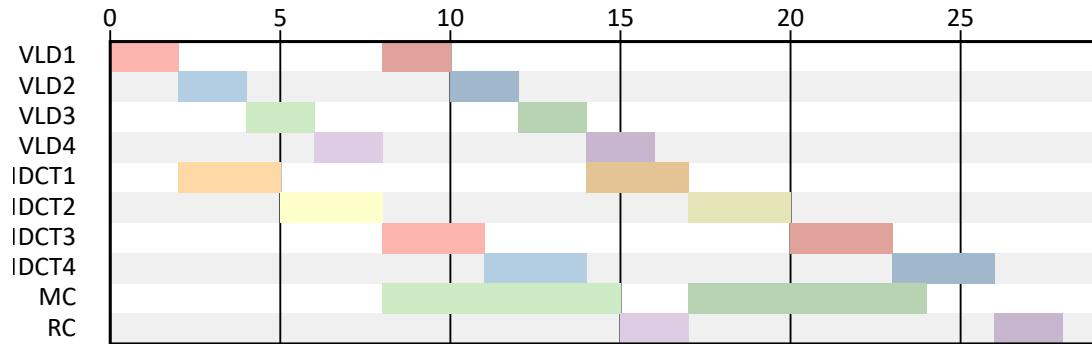
1. The Gantt chart shows two iterations of the following dataflow graph.



2. It has a repetition vector ρ with $\rho(\text{VLD}) = \rho(\text{IDCT}) = 4$ and $\rho(\text{MC}) = \rho(\text{RC}) = 1$.
3. Conversion of the multi-rate dataflow model to a single-rate dataflow graph gives the following result:

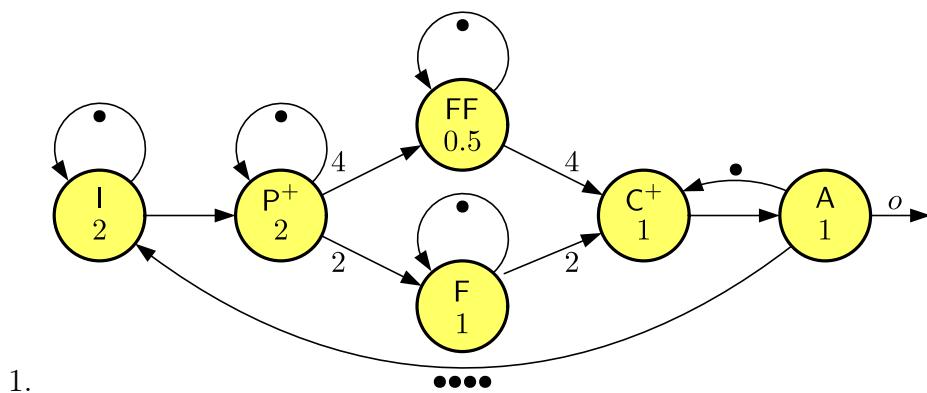


4. Entering this model in the CMWB gives the following Gantt chart.

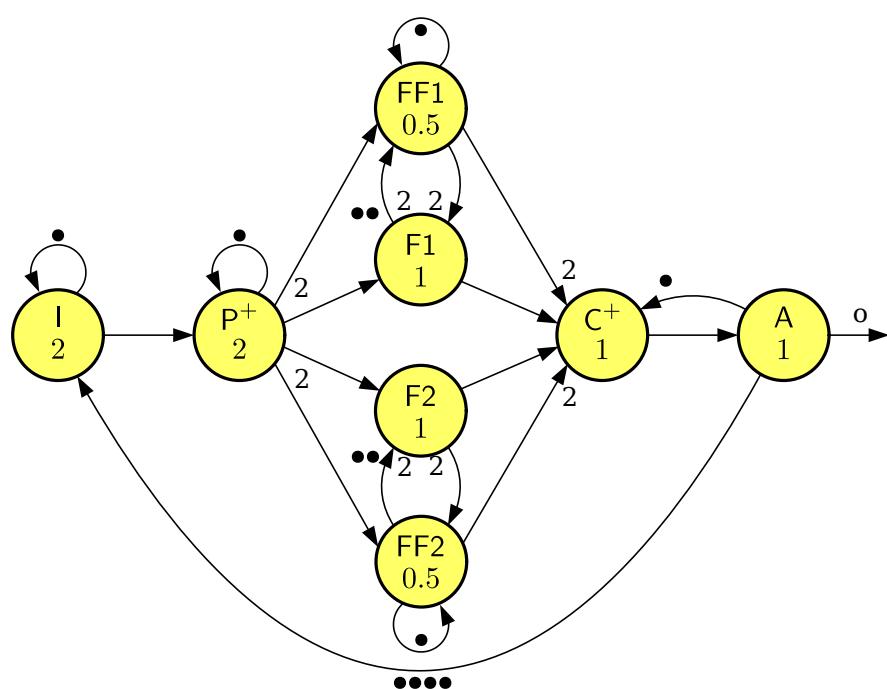
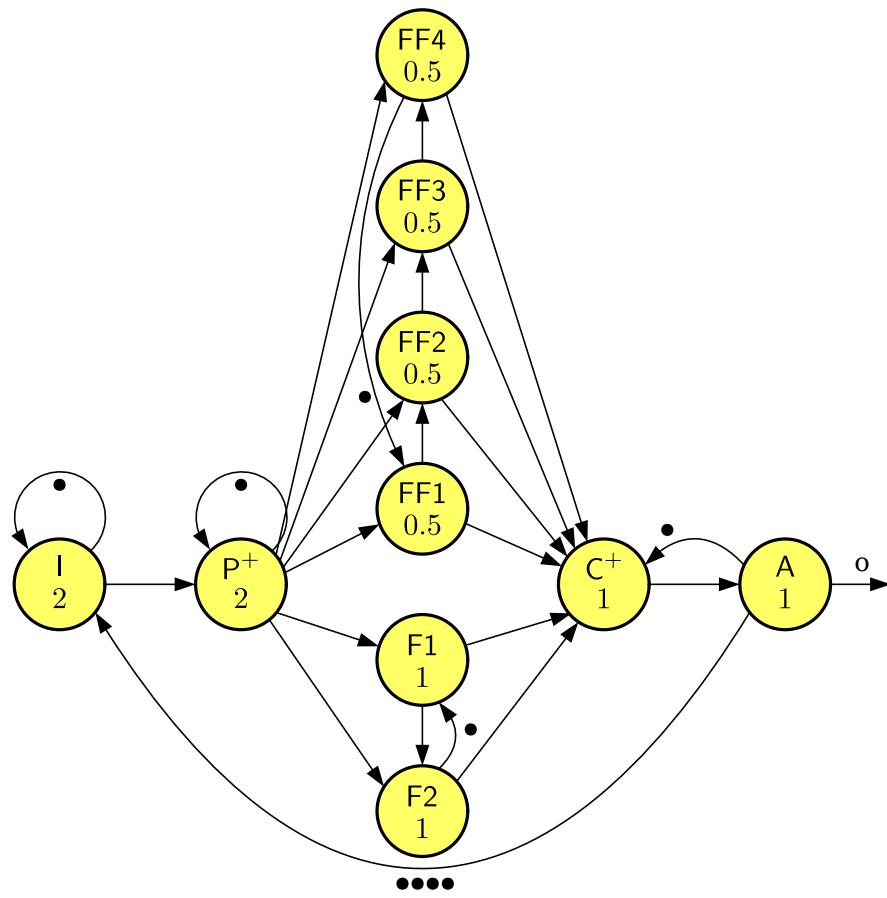


When carefully considering the firings of the four VLD and IDCT actors, one may notice the similarity between the original Gantt chart and this one.

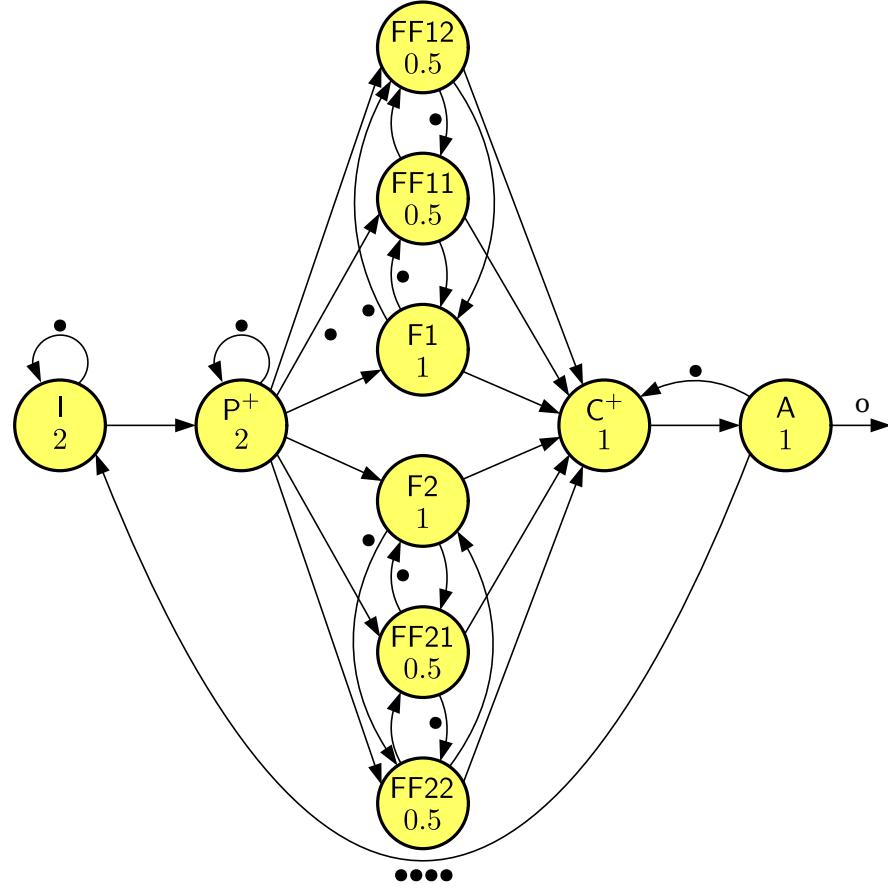
Exercise C.5 (An image-based control system – multi-rate dataflow).



2. Actors I , P^+ , C^+ , and A all have an entry 1 in the repetition vector, F has entry 2, and FF has entry 4.



5. Actors I , P^+ , $F1$, $F2$, C^+ , and A all have entry 1 in the repetition vector, and actors $FF1$ and $FF2$ have entry 2.

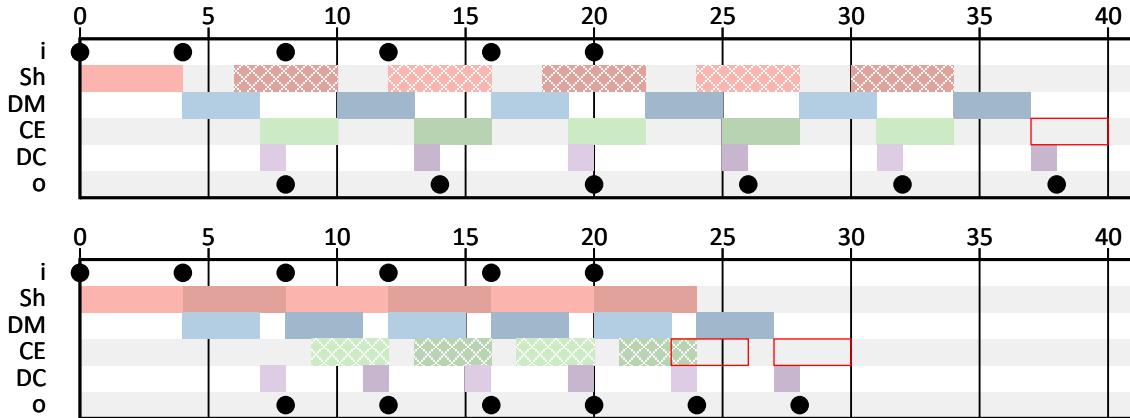


6.

Note that all four models have an actuation rate of 1 actuation per 2 time units and an actuation delay of 8.

Exercise C.6 (A wireless channel decoder – schedules, performance).

1. The ending of the last actor firing of a schedule determines makespan. In both cases, this is the channel estimation actor CE. The makespan is 40 resp. 30 for $n = 1$ and $n = 2$.
2. The latency is the maximal time between the consumption of an input token from the designated input and the production of the corresponding output token on the designated output. For $n = 2$, this duration is constant for all input-output token pairs, namely 8. Hence, the latency is 8. For $n = 1$, this duration grows over time, because the DM-CE actor combination cannot keep up with the input rate. For six iterations, the maximal duration is 18. Hence, for $n = 1$, the latency is 18.
3. For $n = 2$, the schedule is periodic, because it satisfies the condition mentioned in Definition C.2 (Schedules) with period $\mu = 4$; for $n = 1$, the schedule is not periodic. The shift actor Sh fires at a higher rate than the other actors, so there is no period μ that can satisfy the condition of Definition C.2.
4. The following Gantt charts provide alap schedules for $n = 1$ and $n = 2$, respectively.

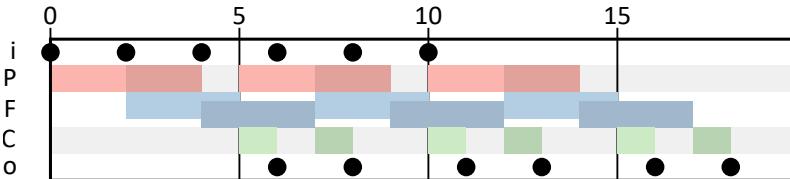


Patterned boxes are actor firings that are delayed with respect to the self-timed schedule. Red-lined boxes illustrate actor firings that are not needed to produce the required output, and should hence be omitted from the schedules. Both schedules are periodic, with periods $\mu = 6$ and $\mu = 4$, respectively.

5. For $n = 1$, the throughput is $1/6$; for $n = 2$, the throughput is $1/4$. Note that the latency between inputs on i and outputs on o grows indefinitely for $n = 1$. Formally, the latency is then not defined; informally, one might say that the latency is infinite.

Exercise C.7 (A producer-consumer pipeline – schedules, performance).

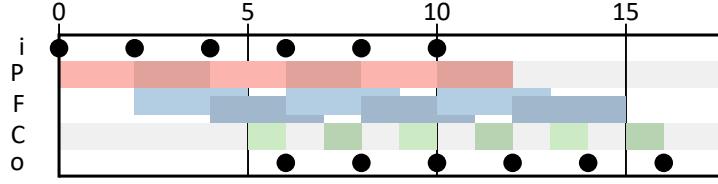
We obtain the following Gantt chart for the first six iterations:



Assume inputs arrive periodically with period 2 and the first input arrives at time 0.

1. The Gantt chart shows that the makespan is 18.
2. The latency grows over time. The maximum value in the schedule for six iterations is 8.
3. No it isn't. None of the actors fires in a periodic pattern.
4. The above self-timed schedule is also an alap schedule for the resulting output production.
5. Actor C fires 2 times per 5 time units, so the throughput is $2/5$.
6. The maximal achievable throughput for Actor C if the buffer capacities may be enlarged is $1/2$. The buffer between P and F needs to be enlarged to size 3; the buffer size between F and C cannot be decreased. We need three spaces in the P-F buffer, because each P

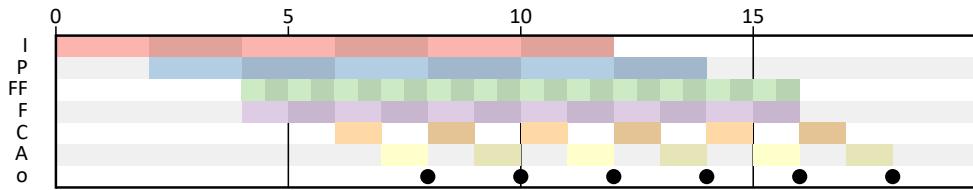
firing claims a space at its start and each F firing only releases a space at its end. The following is a corresponding Gantt chart:



7. The self-timed schedule that maximizes throughput for the minimal buffer capacities of the previous item is periodic with period $\mu = 2$.
8. The self-timed schedule is an alap schedule.

Exercise C.8 (An image-based control system – schedules, performance).

The following is a Gantt chart for the first six iterations:



1. The makespan is 18.
2. The latency is 8. Note that the latency corresponds to the actuation delay in this example.
3. Yes it is. In multi-rate graphs actor firings k and $k + \rho(a)$, where $\rho(a)$ is the repetition-vector entry for actor a , should be at a distance that is equal to the period. In this case, each iteration repeats with period 2.
4. The self-timed schedule given above is an alap schedule.
5. The throughput of Actor A is $1/2$. Throughput corresponds in this example to the actuation rate.

Exercise C.9 (A wireless channel decoder – max-plus analysis).

Let $x_{sh}(k)$, for $k \in \mathbb{N}$, denote the availability time of token k on the self-loop on Actor Sh. By assumption $x_{sh}(0) = 0$. Let $x_{sd}(k)$ and $x_{dd}(k)$ be the token availability times on the Sh-DM and DM-DC channels, respectively.

1. We start with expressions for the four tokens produced by the three individual actor firings: $x_o(k) = 1 \otimes x_{dd}(k)$; $x_{dd}(k) = 3 \otimes (x_{ce}(k) \oplus x_{sd}(k))$; $x_{sd}(k) = x_{sh}(k + 1) = 4 \otimes (x_i(k) \oplus x_{sh}(k))$. Note that these expressions are valid for any $k \in \mathbb{N}$. Substitution

yields $x_o(k) = 1 \otimes 3 \otimes (x_{ce}(k) \oplus x_{sd}(k)) = 4 \otimes (x_{ce}(k) \oplus 4 \otimes (x_i(k) \oplus x_{sh}(k)))$. For $k = 0$, this yields $x_o(0) = 4 \otimes (x_{ce}(0) \oplus 4 \otimes (x_i(0) \oplus x_{sh}(0)))$. Given the assumptions that $x_{sh}(0) = 0$ and that arrival times of tokens on inputs are non-negative, this reduces to $x_o(0) = 4 \otimes (x_{ce}(0) \oplus 4 \otimes x_i(0))$. In the usual notations, this is equal to $x_o(0) = 4 + \max(x_{ce}(0), 4 + x_i(0))$. Taking as an example $x_i(0) = x_{ce}(0) = 0$ gives $x_o(0) = 8$. Note that these assumptions on the inputs and the resulting output time conform to the Gantt chart obtained for the first item of Exercise C.1.

2. From the previous item, we know that $x_o(1) = 4 \otimes (x_{ce}(1) \oplus 4 \otimes (x_i(1) \oplus x_{sh}(1)))$. Substitution in combination with the assumptions yields $x_o(1) = 4 \otimes (x_{ce}(1) \oplus 4 \otimes (x_i(1) \oplus 4 \otimes (x_i(0) \oplus x_{sh}(0)))) = 4 \otimes (x_{ce}(1) \oplus 4 \otimes (x_i(1) \oplus 4))$. In the usual notations, this leads to $x_o(1) = 4 + \max(x_{ce}(1), 4 + \max(x_i(1), 4))$. Taking, in line with the Gantt chart of the first item of Exercise C.1, $x_i(1) = 4$ and $x_{ce}(1) = 10$ leads to $x_o(1) = 4 + 10 = 14$. Also this result is in line with the Gantt chart.
3. If $x_{ce}(k) \leq x_i(k) + 4$, for any $k \in \mathbb{N}$, the production times $x_o(k)$ do not depend on the arrival times $x_{ce}(k)$, because $x_{ce}(k)$ can be eliminated from the expression for $x_o(k)$ obtained in the first item.

Exercise C.10 (A wireless channel decoder – max-plus-linear index-invariant system).

1. *Additivity:* Let $i_1, i_2, ce_1, ce_2, o_1, o_2$, and o_3 be event sequences (with obvious correspondences to inputs and outputs) such that $i_1, ce_1 \xrightarrow{WCD} o_1$, $i_2, ce_2 \xrightarrow{WCD} o_2$, and $i_1 \oplus i_2, ce_1 \oplus ce_2 \xrightarrow{WCD} o_3$. We need to show that $o_3 = o_1 \oplus o_2$.

From the previous exercise, we conclude that $o(k) = 4 \otimes (ce(k) \oplus 4 \otimes (i(k) \oplus (4 \otimes i)^1(k)))$ for any event sequences i, ce, o such that $i, ce \xrightarrow{WCD} o$. Thus, $o_3(k) = 4 \otimes ((ce_1 \oplus ce_2)(k) \oplus 4 \otimes ((i_1 \oplus i_2)(k) \oplus (4 \otimes (i_1 \oplus i_2))^1(k)))$. We distinguish two cases, namely $k = 0$ and $k > 0$, because for these two cases the delay expression $(4 \otimes (i_1 \oplus i_2))^1(k)$ gives different results.

First, assume $k = 0$. Then

$$\begin{aligned}
o_3(0) &= 4 \otimes ((ce_1 \oplus ce_2)(0) \oplus 4 \otimes ((i_1 \oplus i_2)(0) \oplus 4 \otimes -\infty)) \\
&= 4 \otimes ((ce_1 \oplus ce_2)(0) \oplus 4 \otimes ((i_1 \oplus i_2)(0))) \\
&= 4 \otimes (ce_1(0) \oplus ce_2(0) \oplus 4 \otimes i_1(0) \oplus 4 \otimes i_2(0)) \\
&= 4 \otimes ce_1(0) \oplus 4 \otimes ce_2(0) \oplus 8 \otimes i_1(0) \oplus 8 \otimes i_2(0) \\
&= 4 \otimes ce_1(0) \oplus 8 \otimes i_1(0) \oplus 4 \otimes ce_2(0) \oplus 8 \otimes i_2(0) \\
&= \{\text{reversing the reasoning}\} \\
&\quad 4 \otimes (ce_1(0) \oplus 4 \otimes (i_1(0) \oplus 4 \otimes -\infty)) \\
&\quad \oplus 4 \otimes (ce_2(0) \oplus 4 \otimes (i_2(0) \oplus 4 \otimes -\infty)) \\
&= o_1(0) \oplus o_2(0)
\end{aligned}$$

Note that the ‘trick’ in the above reasoning is to rewrite $o_3(0)$ into a sum of products.

Second, assume $k > 0$. Then

$$\begin{aligned}
 o_3(k) &= 4 \otimes ((ce_1 \oplus ce_2)(k) \oplus 4 \otimes ((i_1 \oplus i_2)(k) \oplus (4 \otimes (i_1 \oplus i_2))(k-1))) \\
 &= \{\text{rewrite into a sum of products, similar to the case } k=0\} \\
 &\quad 4 \otimes ce_1(k) \oplus 8 \otimes i_1(k) \oplus 12 \otimes i_1(k-1) \\
 &\quad \oplus 4 \otimes ce_2(k) \oplus 8 \otimes i_2(k) \oplus 12 \otimes i_2(k-1) \\
 &= \{\text{reversing the reasoning}\} \\
 &\quad o_1(k) \oplus o_2(k)
 \end{aligned}$$

This completes the proof of additivity.

2. *Homogeneity*: For any event sequences i, ce, o such that $i, ce \xrightarrow{\text{WCD}} o$ and constant $c \in \mathbb{T}$, we need to show that $c \otimes i, c \otimes ce \xrightarrow{\text{WCD}} c \otimes o$.

We start again from the earlier derived expression for $o(k)$: $o(k) = 4 \otimes (ce(k) \oplus 4 \otimes (i(k) \oplus (4 \otimes i)^1(k)))$. We instantiate it with $c \otimes i$ and $c \otimes ce$, and again distinguish cases $k = 0$ and $k > 0$. Assume applying WCD to $c \otimes i$ and $c \otimes ce$ yields output sequence o' ; that is, $c \otimes i, c \otimes ce \xrightarrow{\text{WCD}} o'$.

First, assume $k = 0$. Then

$$\begin{aligned}
 o'(0) &= 4 \otimes ((c \otimes ce)(0) \oplus 4 \otimes ((c \otimes i)(0) \oplus 4 \otimes -\infty)) \\
 &= (4 \otimes c \otimes ce)(0) \oplus (8 \otimes c \otimes i)(0) \\
 &= c \otimes ((4 \otimes ce)(0) \oplus (8 \otimes i)(0)) \\
 &= c \otimes (4 \otimes (ce(0) \oplus 4 \otimes i(0))) \\
 &= c \otimes (4 \otimes (ce(0) \oplus 4 \otimes (i(0) \oplus 4 \otimes -\infty))) \\
 &= c \otimes o(0)
 \end{aligned}$$

The trick here is to rewrite the expression into a sum of products, then work the multiplication with c to the outermost level, and then reverse the reasoning.

Second, assume $k > 0$. Then, following a similar reasoning as for case $k = 0$, it follows that

$$\begin{aligned}
 o'(k) &= 4 \otimes ((c \otimes ce)(k) \oplus 4 \otimes ((c \otimes i)(k) \oplus (4 \otimes (c \otimes i))(k-1))) \\
 &= (4 \otimes c \otimes ce)(k) \oplus (8 \otimes c \otimes i)(k) \oplus (12 \otimes c \otimes i)(k-1) \\
 &= c \otimes ((4 \otimes ce)(k) \oplus (8 \otimes i)(k) \oplus (12 \otimes i)(k-1)) \\
 &= c \otimes (4 \otimes (ce(k) \oplus 4 \otimes i(k) \oplus 8 \otimes i(k-1))) \\
 &= c \otimes (4 \otimes (ce(k) \oplus 4 \otimes (i(k) \oplus (4 \otimes i)(k-1)))) \\
 &= c \otimes o(k)
 \end{aligned}$$

This proves homogeneity.

3. *Index invariance*: For any event sequences i, ce, o such that $i, ce \xrightarrow{\text{WCD}} o$ and $N \in \mathbb{N}$, we need to show that $i^N, ce^N \xrightarrow{\text{WCD}} o^N$.

Assume $i^N, ce^N \xrightarrow{\text{WCD}} o'$. We start from $o'(k) = 4 \otimes (ce^N(k) \oplus 4 \otimes (i^N(k) \oplus (4 \otimes i^N)^1(k)))$. We distinguish cases $N = 0$, $k < N$, $0 < k = N$, and $k > N$.

For $N = 0$, the result follows immediately because $s^0 = s$ for any event sequence s .

For $k < N$, it follows easily that $o'(k) = -\infty = o^N(k)$.

For $0 < k = N$,

$$\begin{aligned} o'(N) &= 4 \otimes (ce^N(N) \oplus 4 \otimes i^N(N)) \\ &= 4 \otimes (ce(0) \oplus 4 \otimes i(0)) \\ &= o(0) \\ &= o^N(N) \end{aligned}$$

For $k > N$,

$$\begin{aligned} o'(k) &= 4 \otimes (ce^N(k) \oplus 4 \otimes (i^N(k) \oplus (4 \otimes i^N)^1(k))) \\ &= 4 \otimes (ce(k-N) \oplus 4 \otimes (i(k-N) \oplus 4 \otimes i(k-N-1))) \\ &= (4 \otimes (ce \oplus 4 \otimes (i \oplus (4 \otimes i)^1)))(k-N) \\ &= o(k-N) \\ &= o^N(k) \end{aligned}$$

This completes the proof.

Exercise C.11 (The manufacturing system – superposition).

The CMWB can be used to generate the needed output event sequences for all input event sequences of interest. The CMWB can also be used to check the correctness of the outcomes derived through superposition.

1. The production times are then as follows: [16,21,26,33,40].
2. The production times are then as follows: [16,21,28,33,38].
3. The production times can be obtained by taking the element-wise maximum of the results of the previous items: [16,21,28,33,40].
4. The output production times resulting from an input sequence $[5, -\infty, -\infty, -\infty, -\infty]$ are [16,22,27,32,37]. Hence, compared to the production times obtained earlier, the second output will be delayed by 1 time unit.

The output production times resulting from input sequence $[4, -\infty, -\infty, -\infty, -\infty]$ are [16,21,26,31,36]. So the best possible makespan with the first bottom arriving at time 4 is equal to 36. This makespan cannot be further improved. Assuming the unlimited availability of bottoms and tops at time 0 gives the same makespan. (It in fact gives exactly the same production times for all outputs.)

The response to input sequence $[5, -\infty, -\infty, -\infty, -\infty]$ shows that the arrival of the first bottom cannot be further delayed than time 4 without affecting the best possible makespan. The makespan of any schedule in which the first bottom arrives at time 5 cannot be better than 37.

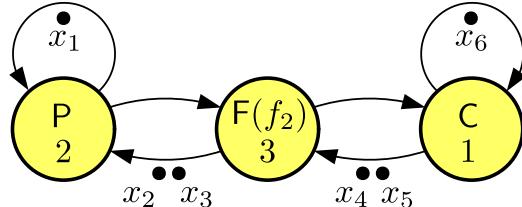
Exercise C.12 (A wireless channel decoder – impulse responses).

1. We need to constrain only the first firing; all other firings automatically get starting times at least 0 because of the dependencies. Therefore, $sh = [0, -\infty, -\infty, -\infty, \dots] = \delta$ is sufficient. $sh = [0, 0, 0, \dots]$ is also possible and leads to the same result, but the following analysis would be more involved.
2. $h_{i,o} = [8, 12, 16, 20, 24, \dots]$, $h_{ce,o} = [4, -\infty, -\infty, -\infty, -\infty, \dots]$, and $h_{sh,o} = [8, 12, 16, 20, 24, \dots]$.
3. Using the generalization of Theorem C.4 to systems with multiple inputs and outputs, we obtain $o = i \otimes h_{i,o} \oplus ce \otimes h_{ce,o} \oplus sh \otimes h_{sh,o} = i \otimes h_{i,o} \oplus ce \otimes h_{ce,o} \oplus \delta \otimes h_{sh,o}$.
4. For o , using the equation of part 3 we obtain with $i = ce = [0, 4, 8, \dots]$

$$\begin{aligned} o(0) &= 0 \otimes 8 \oplus 0 \otimes 4 \oplus 0 \otimes 8 = 8 \\ o(1) &= (i(0) \otimes h_{i,o}(1) \oplus i(1) \otimes h_{i,o}(0)) \oplus (ce(0) \otimes h_{ce,o}(1) \oplus ce(1) \otimes h_{ce,o}(0)) \\ &\quad \oplus (sh(0) \otimes h_{sh,o}(1) \oplus sh(1) \otimes h_{sh,o}(0)) \\ &= (0 \otimes 12 \oplus 4 \otimes 8) \oplus (0 \otimes -\infty \oplus 4 \otimes 4) \oplus (0 \otimes 12 \oplus -\infty \oplus 8) = 12 \\ o(2) &= (0 \otimes 16 \oplus 4 \otimes 12 \oplus 8 \otimes 8) \oplus (0 \otimes -\infty \oplus 4 \otimes -\infty \oplus 8 \otimes 4) \\ &\quad \oplus (0 \otimes 16 \oplus -\infty \otimes 12 \oplus -\infty \otimes 80) = 16 \end{aligned}$$

Exercise C.13 (A producer-consumer pipeline – max-plus matrix equation).

We name the time stamps as in the following picture:



1. We can draw up the following equations:

$$\begin{aligned} x'_1 &= 2 \otimes (x_1 \oplus x_2) = 2 \otimes x_1 \oplus 2 \otimes x_2 \\ x'_2 &= 0 \otimes x_3 \\ x'_3 &= 3 \otimes ((2 \otimes x_1 \oplus 2 \otimes x_2) \oplus x_4) = 5 \otimes x_1 \oplus 5 \otimes x_2 \oplus 3 \otimes x_4 \\ x'_4 &= 0 \otimes x_5 \\ x'_5 &= 1 \otimes ((5 \otimes x_1 \oplus 5 \otimes x_2 \oplus 3 \otimes x_4) \oplus x_6) = 6 \otimes x_1 \oplus 6 \otimes x_2 \oplus 4 \otimes x_4 \oplus 1 \otimes x_6 \\ x'_6 &= x'_5 \end{aligned}$$

These equations give the following max-plus equation:

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \\ x_3(k+1) \\ x_4(k+1) \\ x_5(k+1) \\ x_6(k+1) \end{bmatrix} = \begin{bmatrix} 2 & 2 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ 5 & 5 & -\infty & 3 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ 6 & 6 & -\infty & 4 & -\infty & 1 \\ 6 & 6 & -\infty & 4 & -\infty & 1 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \\ x_3(k) \\ x_4(k) \\ x_5(k) \\ x_6(k) \end{bmatrix}$$

2. Assuming \mathbf{G} denotes the above matrix, we can compute the makespan of the first iteration as $|\mathbf{x}(1)| = |\mathbf{G}\mathbf{0}| = |[2 \ 0 \ 5 \ 0 \ 6 \ 6]^T| = 6$. The makespan of the second iteration is $|\mathbf{x}(2)| = |\mathbf{G}\mathbf{x}(1)| = |[4 \ 5 \ 7 \ 6 \ 8 \ 8]^T| = 8$. Finally, the makespan of the third iteration then is $|\mathbf{G}\mathbf{x}(2)| = |[7 \ 7 \ 10 \ 8 \ 11 \ 11]^T| = 11$.

Exercise C.14 (A producer-consumer pipeline – symbolic simulation).

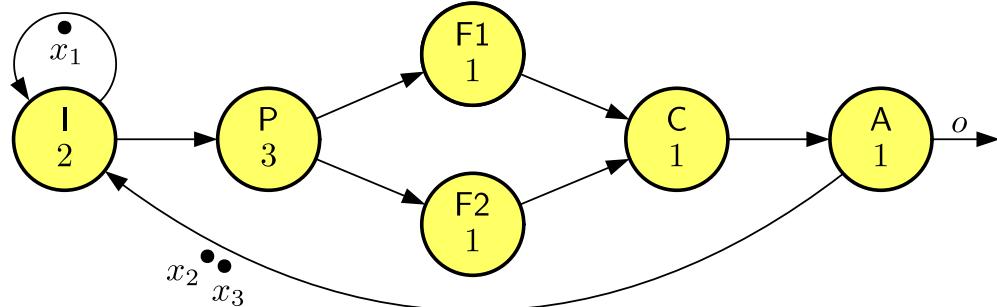
The symbolic simulation can only be performed in one order, starting with P , followed by F , and ending with C :

| actor | symbolic start | symbolic completion |
|-------|---|---|
| P | $[0 \ 0 \ -\infty \ -\infty \ -\infty \ -\infty]^T$ | $[2 \ 2 \ -\infty \ -\infty \ -\infty \ -\infty]^T$ |
| F | $[2 \ 2 \ -\infty \ 0 \ -\infty \ -\infty]^T$ | $[5 \ 5 \ -\infty \ 3 \ -\infty \ -\infty]^T$ |
| C | $[5 \ 5 \ -\infty \ 3 \ -\infty \ 0]^T$ | $[6 \ 6 \ -\infty \ 4 \ -\infty \ 1]^T$ |

The symbolic time-stamp vector for x_1 is the completion time of P ; the symbolic time-stamp vector for x_2 is the initial time-stamp vector for x_3 , \mathbf{i}_3 ; for x_3 , we take the completion time of F , for x_4 , the initial time-stamp vector for x_5 , \mathbf{i}_5 , and for x_5 and x_6 , the completion time of C . The end result is, not surprisingly, the same matrix \mathbf{G} as in the previous exercise.

Exercise C.15 (An image-based control pipeline – symbolic simulation).

We name the time stamps as in the following picture:



1. The symbolic simulation goes as follows:

| actor | symbolic start | symbolic completion |
|-------|-----------------------|-----------------------|
| I | $[0 \ 0 \ -\infty]^T$ | $[2 \ 2 \ -\infty]^T$ |
| P | $[2 \ 2 \ -\infty]^T$ | $[5 \ 5 \ -\infty]^T$ |
| F1 | $[5 \ 5 \ -\infty]^T$ | $[6 \ 6 \ -\infty]^T$ |
| F2 | $[5 \ 5 \ -\infty]^T$ | $[6 \ 6 \ -\infty]^T$ |
| C | $[6 \ 6 \ -\infty]^T$ | $[7 \ 7 \ -\infty]^T$ |
| A | $[7 \ 7 \ -\infty]^T$ | $[8 \ 8 \ -\infty]^T$ |

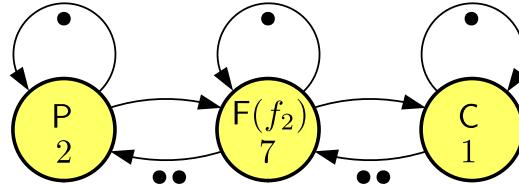
The symbolic time-stamp vector for x_1 is the completion time of \mathbf{l} ; the symbolic time-stamp vector for x_2 is the initial time-stamp vector for x_3 , \mathbf{i}_3 ; for x_3 , we get the completion time of \mathbf{A} . This gives matrix

$$\mathbf{G} = \begin{bmatrix} 2 & 2 & -\infty \\ -\infty & -\infty & 0 \\ 8 & 8 & -\infty \end{bmatrix}$$

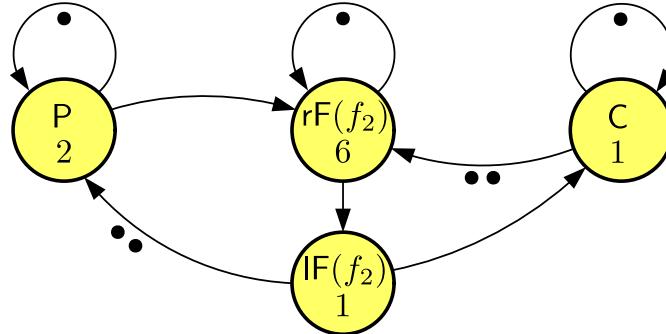
2. To compute the makespan of the first three actuations, we compute $|\mathbf{G}^3 \mathbf{0}|$. The series of time-stamp vectors obtained during the computation is $[2 \ 0 \ 8]^T$, $[4 \ 8 \ 10]^T$, $[10 \ 10 \ 16]^T$. The norm of the last vector, and therefore the makespan of three actuations, is 16. As a next step, we compute $\mathbf{G}^4 \mathbf{0} = \mathbf{G}[10 \ 10 \ 16]^T = [12 \ 16 \ 18]^T$. The fifth sample period starts when both the tokens (corresponding to) x_1 and x_2 have been reproduced after four iterations. That is, the fifth sample period starts at $12 \oplus 16 = 16$. You may check that these results conform to the Gantt chart for this IBC pipeline in Exercise C.3.

Exercise C.16 (A producer-consumer pipeline – worst-case abstractions).

1. The worst-case response time of Actor F is 7, when the pipeline gets two consecutive time slots in the TDM schedule and the data arrives precisely at the end of its turn in the TDM schedule. This can be captured in the model as follows. Note the self-edge that captures the fact that different executions of F cannot overlap when executing on the same processor.



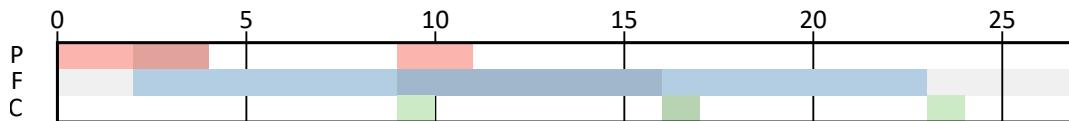
2. Under the mentioned TDM scheduling, Actor F gets 50% of the processing time and has a workload of 3. So the average rate is $1/6$. A worst-case latency of 1 needs to be added to be conservative. Note that F is guaranteed to have 3 time slots in any consecutive 7 time slots in the schedule, so it always completes within 7 time units. This can be modeled as follows.



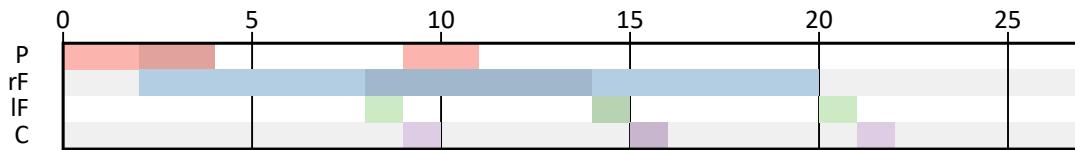
3. The real worst-case conditions occur when the first data arrives for Actor F to process, but its turn in the TDM schedule has just passed. We then get the following Gantt chart, where patterned boxes show that the processing of F has been suspended. The makespan of the third iteration is 22.



The model with the worst-case response time of F as the actor execution time gives the following Gantt chart. The makespan of the third iteration according to this model is 24.



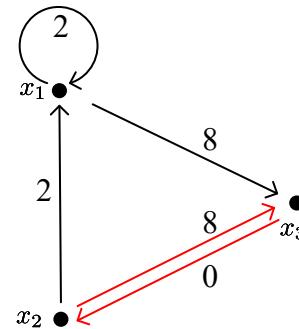
The Gantt chart for the model with a latency-rate model for F is as follows, with a makespan of 22 for three iterations.



Observe that indeed the two models of the first two items are conservative, but they are also both pessimistic. The last model accurately predicts the makespan of (one and) three iterations, but it is pessimistic for the second iteration.

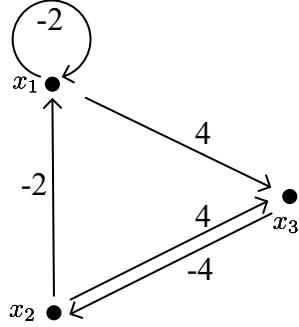
Exercise C.17 (An image-based control system – eigenvalue, eigenvector).

- We obtain the following precedence graph from the matrix computed in Exercise C.15.



The MCM is $8/2 = 4$, derived from the cycle in red. So the eigenvalue is 4.

- We obtain the following precedence graph for the eigenvector by subtracting the eigenvalue 4 from each of the edge weights.



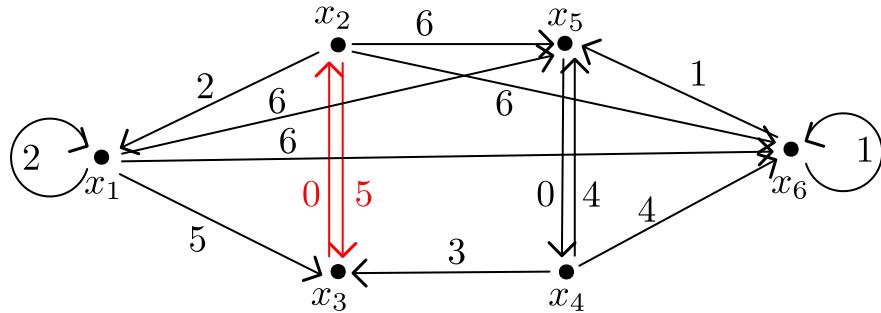
We now select node x_2 on the critical cycle. The longest paths to nodes x_1 , x_2 , and x_3 are then -2 , 0 , and 4 , respectively. This gives normal eigenvector $[-6 \ -4 \ 0]^T$.

- With initial-token time stamps $[0 \ 2 \ 6]^T$, we obtain the following periodic schedule.

$$\begin{cases} \sigma(I, k) = 2 + 4k \\ \sigma(P, k) = 4 + 4k \\ \sigma(F1, k) = \sigma(F2, k) = 7 + 4k \\ \sigma(C, k) = 8 + 4k \\ \sigma(A, k) = 9 + 4k \end{cases}$$

Exercise C.18 (A producer-consumer pipeline – eigenvalue, eigenvector).

- We obtain the following precedence graph from the matrix computed in Exercise C.13.



The MCM is $5/2 = 2.5$, derived from the cycle in red. So the eigenvalue is 2.5 . The critical cycle corresponds to the P–F cycle in the dataflow graph.

- We may obtain a precedence graph for the eigenvector by subtracting the eigenvalue 2.5 from each of the edge weights. Selecting node x_2 on the critical cycle gives eigenvector $[-0.5 \ 0 \ 2.5 \ 1 \ 3.5 \ 3.5]^T$. Normalizing this vector gives $[-4 \ -3.5 \ -1 \ -2.5 \ 0 \ 0]^T$.
- With initial-token time stamps $[0 \ 0.5 \ 3 \ 1.5 \ 4 \ 4]^T$, we obtain the following periodic schedule.

$$\begin{cases} \sigma(P, k) = 0.5 + 2.5k \\ \sigma(F, k) = 2.5 + 2.5k \\ \sigma(C, k) = 5.5 + 2.5k \end{cases}$$

Exercise C.19 (A wireless channel decoder – state-space model).

1. We collect state and input vectors into a single vector as follows: $[\mathbf{x}^T \ i \ ce]^T$ with $\mathbf{x} = [x_1 \ x_2 \ x_3 \ x_4]^T$. The symbolic simulation goes as follows:

| actor | symbolic start | symbolic completion |
|-------|---|---|
| Sh | $[0 \ -\infty \ -\infty \ -\infty \ 0 \ -\infty]^T$ | $[4 \ -\infty \ -\infty \ -\infty \ 4 \ -\infty]^T$ |
| DM | $[4 \ 0 \ 0 \ -\infty \ 4 \ 0]^T$ | $[7 \ 3 \ 3 \ -\infty \ 7 \ 3]^T$ |
| CE | $[7 \ 3 \ 3 \ -\infty \ 7 \ 3]^T$ | $[10 \ 6 \ 6 \ -\infty \ 10 \ 6]^T$ |
| DC | $[7 \ 3 \ 3 \ -\infty \ 7 \ 3]^T$ | $[8 \ 4 \ 4 \ -\infty \ 8 \ 4]^T$ |

The symbolic time-stamp vectors for x_1 and x_2 are the completion times of Sh and DM; the symbolic time-stamp vector for x_3 is the initial time-stamp vector for x_4 ; for x_4 , we get the completion time of CE. For output o , we get the completion time of DC. This gives the following result:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \left[\begin{array}{cccc|cc} 4 & -\infty & -\infty & -\infty & 4 & -\infty \\ 7 & 3 & 3 & -\infty & 7 & 3 \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ 10 & 6 & 6 & -\infty & 10 & 6 \\ \hline 8 & 4 & 4 & -\infty & 8 & 4 \end{array} \right]$$

with

$$\begin{bmatrix} \mathbf{x}(k+1) \\ o(k) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}(k) \\ i(k) \\ ce(k) \end{bmatrix}$$

2. We first take for input i the impulse sequence δ . For ce , we take ϵ , the sequence of only $-\infty$ values. For initial state $\mathbf{x}(0)$, we take ϵ , the vector containing only $-\infty$ as its elements. Using the max-plus representation of the dataflow model, we can then compute the impulse response $\bar{\mathbf{h}}_i$ up to the first four outputs as follows:

$$\bar{\mathbf{h}}_i(0) = \begin{bmatrix} \mathbf{x}(1) \\ o(0) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}(0) \\ i(0) \\ ce(0) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \epsilon \\ 0 \\ -\infty \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ -\infty \\ 10 \\ 8 \end{bmatrix}$$

$$\bar{\mathbf{h}}_i(1) = \begin{bmatrix} \mathbf{x}(2) \\ o(1) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}(1) \\ i(1) \\ ce(1) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} 4 \\ 7 \\ -\infty \\ 10 \\ -\infty \\ -\infty \end{bmatrix} = \begin{bmatrix} 8 \\ 11 \\ 10 \\ 14 \\ 12 \end{bmatrix}$$

$$\bar{\mathbf{h}}_i(2) = \begin{bmatrix} \mathbf{x}(3) \\ o(2) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}(2) \\ i(2) \\ ce(2) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} 8 \\ 11 \\ 10 \\ 14 \\ -\infty \\ -\infty \end{bmatrix} = \begin{bmatrix} 12 \\ 15 \\ 14 \\ 18 \\ 16 \end{bmatrix}$$

$$\bar{\mathbf{h}}_i(3) = \begin{bmatrix} \mathbf{x}(4) \\ o(3) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}(3) \\ i(3) \\ ce(3) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} 12 \\ 15 \\ 14 \\ 18 \\ -\infty \\ -\infty \end{bmatrix} = \begin{bmatrix} 16 \\ 19 \\ 18 \\ 22 \\ 20 \end{bmatrix}$$

It is not difficult to see that $\bar{\mathbf{h}}_i$ proceeds in a periodic pattern with period 4.

In a similar fashion, we can compute the impulse response $\bar{\mathbf{h}}_{ce}$:

$$\bar{\mathbf{h}}_{ce}(0) = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \epsilon \\ -\infty \\ -\infty \\ 0 \end{bmatrix} = \begin{bmatrix} -\infty \\ 3 \\ -\infty \\ 6 \\ 4 \end{bmatrix}, \quad \bar{\mathbf{h}}_{ce}(1) = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} -\infty \\ -\infty \\ 6 \\ -\infty \\ -\infty \end{bmatrix} = \begin{bmatrix} -\infty \\ 3 \\ 6 \\ 6 \\ 9 \\ 7 \end{bmatrix}$$

$$\bar{\mathbf{h}}_{ce}(2) = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} -\infty \\ 6 \\ 6 \\ 9 \\ -\infty \\ -\infty \end{bmatrix} = \begin{bmatrix} -\infty \\ 9 \\ 9 \\ 12 \\ 10 \end{bmatrix}, \quad \bar{\mathbf{h}}_{ce}(3) = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} -\infty \\ 9 \\ 9 \\ 12 \\ -\infty \\ -\infty \end{bmatrix} = \begin{bmatrix} -\infty \\ 12 \\ 12 \\ 15 \\ 13 \end{bmatrix}$$

It is not difficult to see that $\bar{\mathbf{h}}_{ce}$ proceeds in a periodic pattern with period 3.

3. The assumptions imply that we can take for ce the ϵ sequence and for initial state \mathbf{x} the zero vector $\mathbf{0}$. From this we can again compute the outputs with the state-space equations.

$$\begin{bmatrix} \mathbf{x}(1) \\ o(0) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{0} \\ 0 \\ -\infty \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 0 \\ 10 \\ 8 \end{bmatrix}, \quad \begin{bmatrix} \mathbf{x}(2) \\ o(1) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} 4 \\ 7 \\ 0 \\ 10 \\ 4 \\ -\infty \end{bmatrix} = \begin{bmatrix} 8 \\ 11 \\ 10 \\ 14 \\ 12 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{x}(3) \\ o(2) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} 8 \\ 11 \\ 10 \\ 14 \\ 8 \\ -\infty \end{bmatrix} = \begin{bmatrix} 12 \\ 15 \\ 14 \\ 18 \\ 16 \end{bmatrix}, \quad \begin{bmatrix} \mathbf{x}(4) \\ o(3) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} 12 \\ 15 \\ 14 \\ 18 \\ 12 \\ -\infty \end{bmatrix} = \begin{bmatrix} 16 \\ 19 \\ 18 \\ 22 \\ 20 \end{bmatrix}$$

Not surprisingly, self-timed execution follows the impulse response for input i .

4. The first firing of Actor CE produces token $x_4(1)$, which is the third token on the channel to DM. We can analyze the impact of delaying the production of this token through the impulse response for channel ce . From the previous item, we know that $x_4(1) = 10$. So if the execution time of the firing producing this token takes one additional time unit, then it is produced at time 11. We can analyze the effect of this in two steps. First, we constrain the firing of DM that needs this specific token by delaying the impulse response $\bar{\mathbf{h}}_{ce}$ by two samples and a time delay of 11:

$$11 \otimes \bar{\mathbf{h}}_{ce}^2 = \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \end{bmatrix}, \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \end{bmatrix}, \begin{bmatrix} -\infty \\ 14 \\ -\infty \\ -\infty \\ 17 \\ -\infty \end{bmatrix}, \begin{bmatrix} -\infty \\ 17 \\ 17 \\ 20 \\ -\infty \\ -\infty \end{bmatrix}, \dots$$

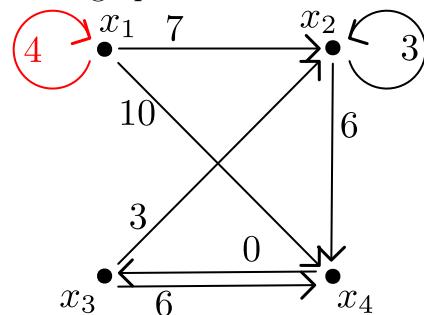
Second, through superposition, we can then compute the effect on the self-timed execution. Assume σ is the schedule computed in the previous item. We then get that

$$\sigma \oplus 11 \otimes \bar{\mathbf{h}}_{ce}^2 = \sigma$$

Hence, the delay in the first firing of Actor CE has no effect on the decoder output.

Exercise C.20 (A wireless channel decoder – throughput).

We obtain the following precedence graph from the \mathbf{A} matrix computed in Exercise C.19.



The MCM is 4, derived from the self-edge in red, corresponding to the self-edge of Actor Sh in the dataflow graph. So the eigenvalue of \mathbf{A} is $\lambda = 4$. From the \mathbf{C} matrix (see Exercise C.19) and the precedence graph, it can be concluded that output o depends on all four state-vector elements. So from Theorem C.5, it follows that the maximal throughput that the decoder may achieve is $1/4$.

Exercise C.21 (An image-based control system – throughput).

In Exercise C.15, we computed matrix \mathbf{G} with symbolic simulation. Since the dataflow graph does not have any inputs, \mathbf{G} equals the \mathbf{A} matrix in the full state-space model. Moreover, the state-space model does not have any \mathbf{B} and \mathbf{D} matrices. Matrix $\mathbf{C} = [8 \ 8 \ -\infty]$ follows immediately from the symbolic simulation of Exercise C.15. The eigenvalue of \mathbf{A} is 4, as derived in Exercise C.17. From the \mathbf{C} matrix and the precedence graph derived in Exercise C.17, it follows that Theorem C.5 applies. Hence, the maximal throughput of the IBC pipeline is $1/4$.

Exercise C.22 (A producer-consumer pipeline – throughput).

1. To compute the state-space model of the graph, we perform symbolic simulation.

| actor | symbolic start | symbolic completion |
|-------|---|---|
| P | $[0 \ 0 \ -\infty \ -\infty \ -\infty \ -\infty \ 0]^T$ | $[2 \ 2 \ -\infty \ -\infty \ -\infty \ -\infty \ 2]^T$ |
| F | $[2 \ 2 \ -\infty \ 0 \ -\infty \ -\infty \ 2]^T$ | $[5 \ 5 \ -\infty \ 3 \ -\infty \ -\infty \ 5]^T$ |
| C | $[5 \ 5 \ -\infty \ 3 \ -\infty \ 0 \ 5]^T$ | $[6 \ 6 \ -\infty \ 4 \ -\infty \ 1 \ 6]^T$ |

This leads to the following state-space model:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \left[\begin{array}{cccccc|c} 2 & 2 & -\infty & -\infty & -\infty & -\infty & 2 \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ 5 & 5 & -\infty & 3 & -\infty & -\infty & 5 \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ 6 & 6 & -\infty & 4 & -\infty & 1 & 6 \\ 6 & 6 & -\infty & 4 & -\infty & 1 & 6 \\ \hline 6 & 6 & -\infty & 4 & -\infty & 1 & 6 \end{array} \right]$$

The \mathbf{A} matrix equals the \mathbf{G} matrix derived in Exercise C.13 for which the eigenvalue, 2.5, was computed in Exercise C.18. From the \mathbf{C} matrix and the precedence graph derived in Exercise C.18, it follows that Theorem C.5 applies. Hence, the maximal throughput of the producer-consumer pipeline is $1/2.5 = 2/5$.

2. The wireless channel decoder of Exercise C.20 has a throughput of $1/4$. The producer-consumer pipeline is slowed down to follow this throughput (Theorem C.6). Thus the throughput of the overall system is $1/4$.

Exercise C.23 (An image-based control system – latency).

The state-space model was derived in Exercise C.21. The \mathbf{A} matrix is the matrix \mathbf{G} of Exercise C.15, the \mathbf{C} matrix is $[8 \ 8 \ -\infty]$. Since the graph has no inputs, the model does not have \mathbf{B} and \mathbf{D} matrices. To compute the desired latency, we only need to compute Λ_{state}^μ

for $\mu = 4$:

$$-\mu \otimes \mathbf{A} = -4 \otimes \begin{bmatrix} 2 & 2 & -\infty \\ -\infty & -\infty & 0 \\ 8 & 8 & -\infty \end{bmatrix} = \begin{bmatrix} -2 & -2 & -\infty \\ -\infty & -\infty & -4 \\ 4 & 4 & -\infty \end{bmatrix}$$

The $*$ -closure $(-\mu \otimes \mathbf{A})^*$ can be computed with the CMWB. It can also be computed as follows.

$$\begin{aligned} (-\mu \otimes \mathbf{A})^* &= (-\mu \otimes \mathbf{A})^0 \oplus (-\mu \otimes \mathbf{A})^1 \oplus (-\mu \otimes \mathbf{A})^2 \\ &= \begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix} \oplus \begin{bmatrix} -2 & -2 & -\infty \\ -\infty & -\infty & -4 \\ 4 & 4 & -\infty \end{bmatrix} \oplus \begin{bmatrix} -4 & -4 & -6 \\ 0 & 0 & -\infty \\ 2 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -2 & -6 \\ 0 & 0 & -4 \\ 4 & 4 & 0 \end{bmatrix} \\ \Lambda_{state}^\mu &= \mathbf{C}(-\mu \otimes \mathbf{A})^* = \begin{bmatrix} 8 & 8 & -\infty \end{bmatrix} \begin{bmatrix} 0 & -2 & -6 \\ 0 & 0 & -4 \\ 4 & 4 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 8 & 4 \end{bmatrix} \end{aligned}$$

So

$$L(o, 4) = \begin{bmatrix} 8 & 8 & 4 \end{bmatrix} \mathbf{0} = 8$$

Exercise C.24 (A wireless channel decoder – latency).

Recall the state-space model of Exercise C.19. We compute the two latency matrices as follows, using the CMWB to compute the $*$ -closure:

$$\begin{aligned} -4 \otimes \mathbf{A} &= \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 3 & -1 & -1 & -\infty \\ -\infty & -\infty & -\infty & -4 \\ 6 & 2 & 2 & -\infty \end{bmatrix} \\ (-\mu \otimes \mathbf{A})^* &= \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 3 & 0 & -1 & -5 \\ 2 & -2 & 0 & -4 \\ 6 & 2 & 2 & 0 \end{bmatrix} \\ \Lambda_{state}^\mu &= \mathbf{C}(-\mu \otimes \mathbf{A})^* = \begin{bmatrix} 8 & 4 & 4 & -\infty \end{bmatrix} \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 3 & 0 & -1 & -5 \\ 2 & -2 & 0 & -4 \\ 6 & 2 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 4 & 4 & 0 \end{bmatrix} \\ \Lambda_{IO}^\mu &= \Lambda_{state}^\mu (-\mu \otimes \mathbf{B}) \oplus \mathbf{D} = \begin{bmatrix} 8 & 4 & 4 & 0 \end{bmatrix} \begin{bmatrix} 0 & -\infty \\ 3 & -1 \\ -\infty & -\infty \\ 6 & 2 \end{bmatrix} \oplus \begin{bmatrix} 8 & 4 \end{bmatrix} = \begin{bmatrix} 8 & 3 \end{bmatrix} \oplus \begin{bmatrix} 8 & 4 \end{bmatrix} = \begin{bmatrix} 8 & 4 \end{bmatrix} \end{aligned}$$

Because the inputs are assumed to be the 4-periodic event sequence, their latency is 0. Since the initial tokens are available at time 0, we can compute the desired latency therefore as follows:

$$L(o, 4) = \begin{bmatrix} 8 & 4 & 4 & 0 \end{bmatrix} \mathbf{0} \oplus \begin{bmatrix} 8 & 4 \end{bmatrix} \mathbf{0} = 8$$

Exercise C.25 (A producer-consumer pipeline – latency).

1. Using the CMWB, we obtain the state-latency and IO-latency matrices:

$$\Lambda_{state}^\mu = \begin{bmatrix} 6 & 6 & 3.5 & 1 & 4 & 1.5 \end{bmatrix}, \Lambda_{IO}^\mu = \begin{bmatrix} 6 \end{bmatrix}$$

So

$$L(o, 2.5) = \begin{bmatrix} 6 & 6 & 3.5 & 1 & 4 & 1.5 \end{bmatrix} \mathbf{0} \oplus \begin{bmatrix} 6 \end{bmatrix} \mathbf{0} = 6$$

2. We know from Exercise C.24 that the latency of the channel decoder is $L(o, 4) = 8$. We can recompute the latency of our producer-consumer pipeline for a period of $\mu = 4$ as

$$\Lambda_{state}^\mu = \begin{bmatrix} 6 & 6 & 2 & 1 & 4 & 0 \end{bmatrix}, \Lambda_{IO}^\mu = \begin{bmatrix} 6 \end{bmatrix}$$

Assuming a latency of 8 for its input from the channel decoder we get:

$$L(o, 4) = \begin{bmatrix} 6 & 6 & 2 & 1 & 4 & 0 \end{bmatrix} \mathbf{0} \oplus \begin{bmatrix} 6 \end{bmatrix} \begin{bmatrix} 8 \end{bmatrix} = 14$$

Thus, the combined system has a latency of 14. (Note that, in this case, the latency of the combined system is the sum of the latencies of the two constituent systems. This is not in general true. You may try to construct an example.)

Exercise C.26 (A wireless channel decoder – stability).

Recall the state-space model of Exercise C.19.

1. No, this self-timed schedule is not stable. The self-timed schedule produces an output $o(k)$ at times $8 + 4k$, for all k . We can consider a simple disturbance of a delay of 1 on the initial input on i . That is, we assume input $1 \otimes \delta$ for i and determine its effect on the self-timed schedule through superposition. In Exercise C.19, we computed the impulse response $\bar{\mathbf{h}}_i$. From this, we know that $o(k)$ occurs at times $8 + 4k$ in this impulse response. So when delaying the impulse input with 1 time unit, to $1 \otimes \delta$, $o(k)$ occurs at times $9 + 4k$. Through superposition with production times $8 + 4k$, we know that $o(k)$ also occurs at times $9 + 4k$ in the self-timed schedule with a delayed first input. Hence, all outputs are delayed by 1 time unit, showing instability.
2. Yes, the schedule is stable. From the impulse response $\bar{\mathbf{h}}_{ce}$ computed in Exercise C.19 (which is also valid for the adapted model; why?), we know that $o(k)$ occurs at times $4 + 3k$ in this impulse response. So if we delay any input m of the ce input by an arbitrary constant c , then we know that $o(k)$ for $k \geq m$ occurs at $4 + c + 3k$ in the delayed impulse response. Through superposition, and since $4 + c + 3k \leq 4 + 4k$ for any

$k \geq m + c$, it follows that the disturbance dissolves in the self-timed schedule after (at most) c outputs. Hence, the self-timed schedule is stable. (Observe that this example illustrates that a schedule may be stable even if it requires the maximal sustainable throughput. The reason is that the actor that limits the throughput does not depend on the input.)

Appendices

After studying these appendices, the student should be able to

- **read** and **understand** basic concepts and notations from set theory;
- **understand** the notions of relation and function;
- **use** these notations **to precisely specify** sets, functions, and relations.

I Sets

Sets and their elements

A **set** is an object that is a collection of other objects, called the *elements* of the set.

Example I.1 (Sets). Small sets can be defined by enumerating all elements. For example, the set of all odd natural numbers^a less than 10 can be written down as follows:

$$\{1, 3, 5, 7, 9\}.$$

Curly braces are the standard notation for denoting sets. Large sets cannot be enumerated conveniently and infinite sets cannot be enumerated at all. In such cases, the following notation can be used:

$$\{n \in \mathbb{N} \mid n \text{ is odd}\}.$$

This set denotes the collection of all odd natural numbers.

^aThe set of natural numbers \mathbb{N} contains all integer numbers from 0 onwards, i.e., 0, 1, 2,

In general, a set of elements taken from some *domain* D can be defined as follows:

$$\{x \in D \mid P(x)\},$$

where P is some predicate on elements of the domain D . A predicate specifies properties of elements from the domain, and can be used to restrict the elements of a set to those of

interest. In the example of the odd natural numbers, the domain is the set \mathbb{N} of natural numbers, and the predicate is the oddness of a natural number.

The above set of all odd natural numbers less than 10 can also be defined as follows:

$$\{n \in \mathbb{N} \mid n \text{ is odd} \wedge n < 10\},$$

where \wedge is the logical notation for ‘and’.

An object s is an element of the set $\{x \in D \mid P(x)\}$, denoted $s \in \{x \in D \mid P(x)\}$, if and only if s is an element of D and predicate P holds for s . That is,

$$s \in \{x \in D \mid P(x)\} \text{ if and only if } s \in D \text{ and } P(s).$$

Note that we have been using the \in notation already in the general notation for defining sets, namely to specify the domain from which elements of a set are taken. Since a domain is also a set, this use of notation is consistent. The number of elements of some set A is often denoted $|A|$. To express that s is not an element of some set A , the notation \notin is used.

$$s \notin A \text{ if and only if not } s \in A.$$

The empty set

The special set without any elements is denoted \emptyset .

Subsets A set can be a part of another set. A set A is a *subset* of some other set B , written as $A \subseteq B$, if and only if all elements of A are also elements of B .

$A \subseteq B$ if and only if, for all $x \in A$, it holds that $x \in B$.

Set A is a *strict subset* of B , $A \subset B$, if and only if all elements of A are also elements of B but in addition B has elements that are not part of A .

$$A \subset B \text{ if and only if } A \subseteq B \text{ and } A \neq B.$$

As a straightforward example, the set of all odd natural numbers less than 10 is a strict subset of the set of all odd natural numbers.

Equality of sets Two sets are equal if and only if they contain precisely the same elements. For example, the set of positive integer numbers is the same as the set of positive natural numbers, and also the sets $\{1, 3, 5, 7, 9\}$ and $\{n \in \mathbb{N} \mid n \text{ is odd} \wedge n < 10\}$ are equal. Using the notion of a subset, equality of two sets A and B can conveniently be defined as follows:

$$A = B \text{ if and only if } A \subseteq B \text{ and } B \subseteq A.$$

Complement Often, it is convenient to assume some superset U of all objects of interest, called the *universe*. Every set of interest is then a subset of this universe. Let A be some set of elements from U , i.e., $A \subseteq U$. The *complement* of A , denoted \bar{A} , is then the set of all elements of U that are not elements of A .

$$\bar{A} = \{x \in U \mid x \notin A\}.$$

Example I.2 (Complement). Assume the universe of elements is the set of natural numbers. Two simple examples of the complement operation are the following: $\{n \in \mathbb{N} \mid n \text{ is odd}\} = \{n \in \mathbb{N} \mid n \text{ is even}\}$ and $\emptyset = \mathbb{N}$.

Note that the complement of a set may change if the universe changes. Assume, for example, that the universe U is the set of natural numbers less than 10. Then, $\emptyset = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

An important property of complement is the following: for any set A , $\bar{\bar{A}} = A$.

Intersection and union Let A and B be sets of elements from universe U . The *intersection* of A and B , denoted $A \cap B$, is the set that contains all elements of U that are contained in *both* A and B . The *union* of A and B , denoted $A \cup B$, is the set that contains all elements of U that are contained in *either* A or B .

$$A \cap B = \{x \in U \mid x \in A \wedge x \in B\} \text{ and}$$

$$A \cup B = \{x \in U \mid x \in A \vee x \in B\},$$

where \vee is the logical notation for ‘or’.

Example I.3 (Set intersection, union).

$$\begin{aligned} \{n \in \mathbb{N} \mid n > 5\} \cap \{n \in \mathbb{N} \mid n < 10\} &= \{6, 7, 8, 9\}, \\ \{n \in \mathbb{N} \mid n > 5\} \cap \{n \in \mathbb{N} \mid n > 10\} &= \{n \in \mathbb{N} \mid n > 10\}, \\ \{n \in \mathbb{N} \mid n \text{ is odd}\} \cap \{n \in \mathbb{N} \mid n \text{ is even}\} &= \emptyset, \text{ and} \\ \{n \in \mathbb{N} \mid n > 5\} \cup \{n \in \mathbb{N} \mid n < 10\} &= \mathbb{N}, \\ \{n \in \mathbb{N} \mid n > 5\} \cup \{n \in \mathbb{N} \mid n > 10\} &= \{n \in \mathbb{N} \mid n > 5\}, \\ \{n \in \mathbb{N} \mid n \text{ is odd}\} \cup \{n \in \mathbb{N} \mid n \text{ is even}\} &= \mathbb{N}. \end{aligned}$$

Some properties of intersection and union are the following (with A, B, C sets):

$$\begin{aligned} A \cap B &= B \cap A, \\ A \cup B &= B \cup A, \\ (A \cap B) \cap C &= A \cap (B \cap C), \\ (A \cup B) \cup C &= A \cup (B \cup C), \\ \overline{A \cap B} &= \bar{A} \cup \bar{B}, \text{ and} \\ \overline{A \cup B} &= \bar{A} \cap \bar{B}. \end{aligned}$$

The first two properties are so-called *commutativity* properties; the next two properties are *associativity* properties; the last two properties resemble the De Morgan laws known from logic. Associativity of an operation implies that we can omit parentheses in expressions like $(A \cap B) \cap C$, simplifying the expression to $A \cap B \cap C$. If an operation is both commutative and associative, performing the operation on any number of operands in an arbitrary order always gives the same result. For example, taking the union of seven different sets, always gives the same result independent of the order in which these seven sets are put together.

Set difference Let A and B be sets of elements from universe U . The *difference* of A and B , denoted $A \setminus B$, is the set that contains all elements of A *except* those elements that are contained in B as well.

$$A \setminus B = \{x \in U \mid x \in A \wedge x \notin B\}.$$

Example I.4 (Set difference).

$$\{n \in \mathbb{N} \mid n > 5\} \setminus \{n \in \mathbb{N} \mid n < 10\} = \{n \in \mathbb{N} \mid n \geq 10\} \text{ and}$$

$$\{n \in \mathbb{N} \mid n > 5\} \setminus \{n \in \mathbb{N} \mid n > 10\} = \{6, 7, 8, 9, 10\}.$$

Note that the notion of set difference provides an alternative way to define complement:

$$\bar{A} = U \setminus A.$$

Power sets Every set has subsets, even the empty set which has itself as a subset. It can sometimes be interesting to consider the set of all subsets of some given set A with elements from universe U . This set is called the power set of A and is often denoted 2^A .

$$2^A = \{B \subseteq U \mid B \subseteq A\}.$$

Example I.5 (Power set).

$$2^{\{1,2,3\}} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\} \text{ and}$$

$$2^\emptyset = \{\emptyset\}.$$

The notation for the power set is derived from the fact that the number of elements of the power set of a finite set A is $2^{|A|}$, i.e., two to the power $|A|$.

Cartesian products If A and B are sets, then the *Cartesian product* $A \times B$ is the set that consists of all *pairs* (a, b) with a an element of A and b an element of B :

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}.$$

Example I.6 (Cartesian product).

$$\mathbb{N} \times \mathbb{N} = \{(n, m) \mid n \in \mathbb{N} \wedge m \in \mathbb{N}\}, \text{ i.e., } \mathbb{N} \times \mathbb{N} \text{ contains all pairs of natural numbers.}$$

Similarly, $A \times B \times C$ contains all triples from elements of sets A , B , and C ; in general, the Cartesian product $A_1 \times \dots \times A_n$, with A_1 through A_n sets, contains all n -tuples (a_1, \dots, a_n) with $a_i \in A_i$. Note that the order of the elements in a tuple is important. For example, the pair $(3, 5)$ differs from the pair $(5, 3)$. (In contrast, the set $\{3, 5\}$ is the same as the set $\{5, 3\}$.) Further note that a Cartesian product of identical sets is sometimes written as a power: e.g., $A \times A$ may be written as A^2 , $A^3 = A \times A \times A$, and $A \times B \times B$ can be written as $A \times B^2$.

Cardinality If A is a set, then $|A|$ is the *cardinality* of the set, defined as the number of elements of the set.

Example I.7 (Cardinality).

- $|\{6, 7, 8, 9\}| = 4$.
- $|2^{\{1,2,3\}}| = 8$.
- $|\mathbb{N}| = \omega$; see the intermezzo on infinity below.
- $|\{n \in \mathbb{N} \mid n \geq 10\}| = \omega$.

Although it may seem counterintuitive, any infinite subset of natural numbers has the same cardinality.

Intermezzo - Infinity

The set \mathbb{N} of natural numbers contains infinitely many different numbers. However, it does not contain infinity itself. To reason about infinity, we introduce a separate symbol for it.

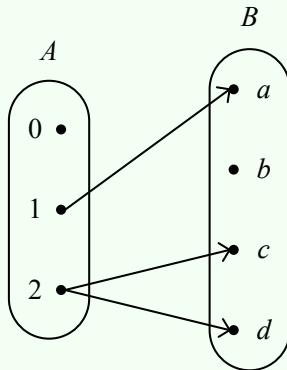
Infinity is denoted by ω .

Formally, ω is the cardinality of the set of natural numbers (meaning it is the first infinite ordinal). We may extend the set \mathbb{N} of natural numbers with ω , to $\mathbb{N} \cup \{\omega\}$. Addition and ‘less than’ can then be defined on this extended set as follows. For every natural number $n \in \mathbb{N}$, it holds that $n < \omega$, i.e., infinity is larger than any natural number. Furthermore, for all $n \in \mathbb{N}$, $\omega + n = n + \omega = \omega$, and $\omega - n = \omega$. The idea is that adding something to infinity or subtracting something from it, keeps the result infinite. Finally, $\omega + \omega = \omega$.

II Relations and functions

A ***relation*** is a mathematical structure that relates elements from one set to elements of another set.

Example II.1 (Relation). Let A be the set $\{0, 1, 2\}$ and B the set $\{a, b, c, d\}$. Suppose that relation R relates element 1 to a and 2 to both c and d . Graphically, this can be represented as follows:



Relation R is a subset of the Cartesian product $A \times B$ defined as follows: $R = \{(1, a), (2, c), (2, d)\}$. The set contains precisely those pairs that are related.

A relation such as R is sometimes called a *binary* relation to emphasize that it relates elements from precisely two sets. In general, it is possible to define relations that relate elements of an arbitrary number of sets.

If R is a relation from A to B , i.e., $R \subseteq A \times B$, then A is said to be the *domain* of R and B is the *codomain* of R . If the domain and codomain of a relation coincide, say they are both equal to some set X , then this relation is said to be a relation *on* X .

We are all familiar with many relations. ‘Being married to’ is an example of a relation on the set of humans. Examples of binary relations on the natural numbers are $<$, $+$, and $=$. Relation $< \subseteq \mathbb{N} \times \mathbb{N}$ contains for example the pair $(3, 4)$ but not the pairs $(3, 3)$ or $(4, 3)$. Note that it is common practice to write $3 < 4$ instead of $(3, 4) \in <$ to denote that 3 is related to 4 by $<$. This so-called *infix* notation is for many binary relations often preferred over the element notation.

A ***function*** is a relation for which every element from the domain is related to *at most one* element from the codomain.

Notation $f : A \rightarrow B$ specifies that f is a function from domain A to codomain B .

Example II.2 (Function). An example of a function is the square function on natural numbers, $f : \mathbb{N} \rightarrow \mathbb{N}$ with $f(x) = x^2$.

Since any function is also a relation, it is possible to see f as a set of pairs: $f \subseteq \mathbb{N} \times \mathbb{N}$ with $f = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid y = x^2\}$. Not every relation is a function. The example relation

R discussed above is not a function because element 2 is related to two elements from the codomain.

A function for which *every* element of the domain is mapped onto an element of the codomain is called a ***total function***; it is called ***partial*** otherwise.

Bibliography

- [1] H. Alizadeh Ara, A. Behrouzian, M. Geilen, M. Hendriks, D. Goswami, and T. Basten. Analysis and visualization of execution traces of dataflow applications. In *Proc. IDEA 2016: Integrating Dataflow, Embedded Computing, and Architecture*. Report ESR-2017-01, pages 19–20. Eindhoven University of Technology, Dept. of Elec. Eng., the Netherlands, 2017.
- [2] F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and linearity: an algebra for discrete event systems*. John Wiley & Sons Ltd, 1992.
- [3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT press, Cambridge, MA, USA, 2008.
- [4] T. Basten, J. Bastos, R. Medina, B. van der Sanden, M.C.W. Geilen, D. Goswami, M.A. Reniers, S. Stuijk, and J.P.M. Voeten. Scenarios in the design of flexible manufacturing systems. In *System-Scenario-based Design Principles and Applications*, pages 181–224. Springer, 2020.
- [5] S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, and J. Takala. *Handbook of signal processing systems*. Springer, 2018.
- [6] K.L. Chung. *Markov Chains with Stationary Transition Probabilities*. Springer, 1967.
- [7] D.R. Cox and H.D. Miller. *The Theory of Stochastic Processes*. Springer Texts in Statistics. CRC Press, 1977.
- [8] B. De Schutter and T. van den Boom. Max-plus algebra and max-plus linear discrete event systems: An introduction. In *Proc. 9th Int. Workshop on Discrete Event Systems, WODES’08*, pages 36–42, Göteborg, Sweden, May 2008.
- [9] B. De Schutter, T. van den Boom, J. Xu, and S.S. Farahani. Analysis and control of max-plus linear discrete-event systems: An introduction. *Discrete Event Dynamic Systems: Theory and Applications*, 30:25–54, 2020.
- [10] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer, Berlin, Germany, 1990.
- [11] K. Etessami. Stutter-Invariant Languages, ω -Automata, and Temporal Logic. In *Proc. Int. Conf. on Computer Aided Verification, CAV 99*, volume 1633 of *LNCS*, pages 236–248. Springer, 1999.

- [12] M.C.W. Geilen, M. Skelin, J.R. van Kampenhout, H. Alizadeh Ara, T. Basten, S. Stuijk, and K.G.W. Goossens. Scenarios in dataflow modeling and analysis. In *System-Scenario-based Design Principles and Applications*, pages 145–180. Springer, 2020.
- [13] R.M.P. Goverde. Railway timetable stability analysis using max-plus system theory. *Transportation Research Part B: Methodological*, 41(2):179–201, 2007.
- [14] B. Heidergott, G.J. Olsder, and J. Van Der Woude. *Max Plus at work: modeling and analysis of synchronized systems: a course on Max-Plus algebra and its applications*. Princeton University Press, 2014.
- [15] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [16] A.F. Karr. *Probability*. Springer Texts in Statistics. Springer, 1993.
- [17] A. Lele, O. Moreira, and P.J.L. Cuijpers. A new data flow analysis model for TDM. In *Proc. 10th ACM Int. Conf. on Embedded Software, EMSOFT 2012*, pages 237–246. ACM, 2012.
- [18] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.
- [19] P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, Sudbury, MA, USA, 4th edition, 2006.
- [20] O. Moreira. *Temporal Analysis and Scheduling of Hard Real-Time Radios running on a Multi-Processor*. PhD thesis, Eindhoven University of Technology, January 2012.
- [21] J.R. Norris. *Markov Chains*. Cambridge University Press. MIT, 2012.
- [22] POOSL. <https://poosl.esi.nl/>.
- [23] N. Privault. *Understanding Markov Chains, Examples and Applications*. Springer Undergraduate Mathematics Series. Springer, 2018.
- [24] W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B: Formal Models and Semantics, pages 133–191. Elsevier, Amsterdam, The Netherlands, 1990.